APL
MANAGEMENT PROBLEMS with ANSWERS and "KIT OF TOOLS"

B. Legrand
Ingenieur de l'Ecole Centrale des Arts et Manufactures

Translated by
Julian Glyn Matthews, Linguatech

John Wiley & Sons Ltd
Chichester · New York · Brisbane · Toronto · Singapore

First published 1982 © Masson, Editeur, Paris under the title APL: Problemes de gestion corriges et B[illegible]tils

Drawings by RUMWEISS

Library of Congress Cataloging in Publication Data:

Legrand, B.
APL, management problems with answers and "kit of tools."
Translation of: APL, problèmes de gestion corrigés et boîtes à outils.
1. APL (Computer program language—Problems, exercises, etc. I. Title. II. Title: A.P.L., management problems with answers and "kit of tools."
QA76.73.A27L4313 1984   001.64'24   83-25908
ISBN 0 471 90334 5

British Library Cataloguing in Publication Data:

Legrand, B.
APL: management problems, with answers and kit of tools.
1. APL (Computer program language)
I. Title
001.64'24   QA76.73.A27

ISBN 0 471 90334 5

Printed in Great Britain

Contents

	Problems	Answers
Important Points	10	
Preliminary Work	11	83
Settlement of Accounts	15	86
Five Columns into One	17	90
Selected Invoices	20	94
Now, Throw Away My Book	23	98
"Hope" and "Truth"	25	100
Everything is in the Presentation	27	103
Monte-Carlo Method	29	104
The Most Useful Functions	31	106
Searching for Skilled Welders	33	109
Will it be Fine Tomorrow?	38	113
Crossed Numbering	41	117
A Difficult Choice	44	121
The Carrot and the Stick	47	125
In the Time of the Pyramids	49	127
Flash	51	129
Week-End Sailing at Brighton	54	131
Can You Update?	59	139
Special Printing	61	142
The Client	63	146
Block-Heads	70	154
Oil on Troubled Waters	71	156
Three Puzzles	77	164
Table of Functions	167	

Foreword

Over ten years, when promoting and teaching APL to the widest public, I have frequently stated that it is not sufficient simply to know the language in order to make good use of it.
Too often I have met people, taught in haste or incompletely, or by clumsy programmed teaching, who fully in good faith have written cumbersome programs with rigid instructions. Such styles of writing gravely compromise possibilities of subsequent development, make it difficult to maintain any position and seriously harm APL's image.

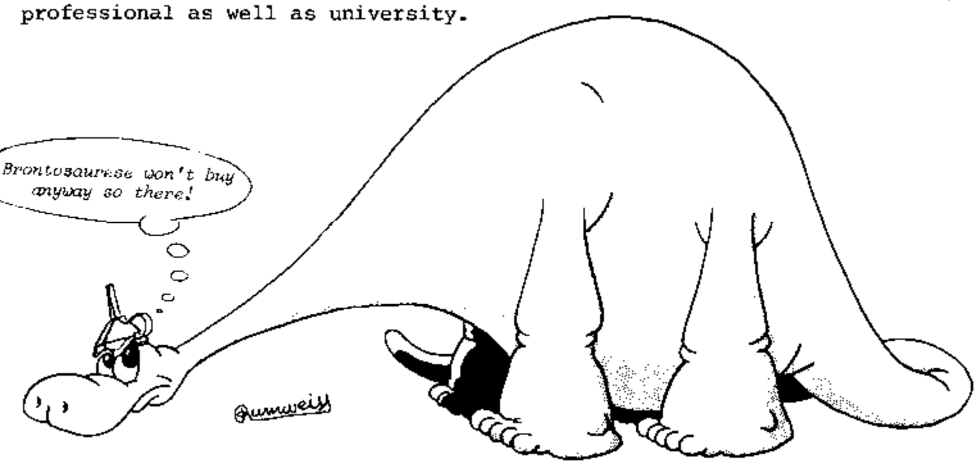
May the following pages help to demonstrate that simple, short, clear programs which can be maintained may be created which are at the same time more efficient than "brontasaurus" programs, which are all to often encountered and which are even sometimes sold by computer manufacturers.

This work is intended equally for those wishing to study APL in depth, and for those who have the task of teaching it properly. For them, I have deliberately chosen themes which are useful in the field of advanced application of APL, namely assistance in decision making.
The solutions presented cover a wide range of classic APL methods.

The answers should also interest APL users wishing to equip themselves with an excellent "Kit of Tools" for solving everyday problems.

Even though there is some progression in complexity of the subjects presented, right from the start the solutions call upon the full richness of the language, without exception. It is therefore advisable to start reading this work only after having acquired an adequate basis in the whole language, or with guidance from someone with considerable experience of APL.
I have purposely excluded any subject requiring the use of files, on the one hand because of the systems offered by manufacturers, and on the other hand because the use of files does not significantly qualify either intellectual progression or the general programming principles.

I hope that this book will promote the advancement of APL in all sectors, professional as well as university.



PROBLEMS

IMPORTANT POINTS

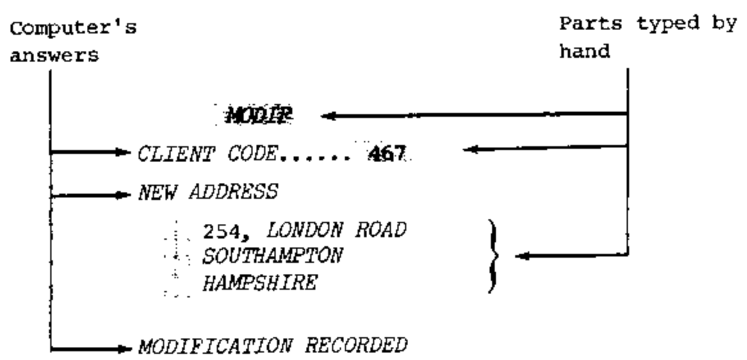
It is difficult to attribute a level of difficulty to a subject, and the following order is very subjective. In any event, start by solving the subjects which feature under heading "Preliminary work", as these are essential for understanding what follows.
Afterwards do not hesitate to tackle the subjects in the order which suits you, according to your interest in the themes.

Most of the subjects are divided into stages and this is the best method for readily solving sometimes complex problems. Abide by these stages, because each presents a classic procedural step.
Solve each subject completely, even though in order to achieve this, you may have to consult the answers between the stages.

Although you may have completely solved a subject, consult the answer. The method represented may be different from yours; comparison can only be beneficial.

Some subjects, and numerous answers make a reference to the methods studied in a course book. This is the work *"Learning and Applying the APL Language"*, by the same author.

To facilitate reading of the processing examples, the parts typed by hand are printed on a grey strip, while the computers answers are printed normally:



PRELIMINARY WORK

Some simple functions can facilitate expression of everyday tasks in APL, and consequently ease writing and maintenance of programs.
Each APL user has his own techniques, and his own set of functions, which he affectionately calls his "*Kit of Tools*".

Here are some functions which I frequently use. They are very simple and extremely useful.

For the remainder of this work, we will assume that these functions are known. They will be used in statements as well as in answers, so as to facilitate programming.

Of course these initial themes to be considered represent only a starter.

FIRST THEME

It is quite simple to designate a series of numbers by writing, for example:

345 to 361

The same flexibility can be obtained in APL, by writing a dyadic function called *TO*, proceeding as follows:

```
      67 TO 82
67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82
```

You can already write such a function.

It can however be generalised. If we wish to designate a series of values some of which are consecutive, we can use the following formula:

41 52, 56 to 65, 77 83 98, 105 to 112, 123 134

This expression presents the following series of numbers:

41 52 56 57 58 59 60 61 62 63 64 65 77 83 98 105 106 107 108 109,
110 111 112 123 and 134

Can you generalise your function *TO* so as to provide a similar expression:

21 32 *TO* 36 44 51 59 *TO* 62 87 88 91

21 32 33 34 35 36 44 51 59 60 61 62 87 88 91

SECOND THEME

In a program it is common practice to print a message and then jump to another instruction.
For example:

```
[5] '----> ABNORMAL SHAPE'
[6] →AGAIN
```

These two instructions can be reduced to a single expression:

```
[5] →AGAIN AFTER 'ABNORMAL SHAPE'
```

Write this auxiliary function, called *AFTER*.

THIRD THEME

We often have to print an array of characters and a numeric vector facing each other.

For example, here is the array *DWARFS*, which contains a list of names:

```
DOC
SNEEZY
BASHFUL
GRUMPY
DOPEY
SLEEPY
HAPPY
```

and here is the vector *DIAMS*, which contains outputs of precious stones:

```
308 144 506 509 481 321 624
```

We wish to print this information in the form of an array, numbering the lines, as follows:

```
1 DOC        308
2 SNEEZY     144
3 BASHFUL    506
4 GRUMPY     509
5 DOPEY      481
6 SLEEPY     321
7 HAPPY      624
```

This requires the vectors to be transformed into characters, and placed vertically. A function *VERT* will serve to carry out this conversion. It will accept the printing format of the vector as left-hand argument.

For example:

```
      6 0 VERT 34 71 89 30
    34
    71
    89
    23
    30
```

or again:

```
      6 2 VERT  44.5  67.32  80  29.175

 44.50
 67.32
 80.00
 29.18
```

Write this function *VERT*.

The presentation sought would be obtained by:

```
      (1 0 VERT ιρDIAMS),' ',DWARFS, (5 0 VERT DIAMS)
```

FOURTH THEME

However, it appears that the output of precious stones has considerably increased as compared with last week. This production has been registered in the vector *MIDAS*:

```
      292 100 402 477 388 359 535
```

It is tempting to calculate the percentage output increase from one week to the next.

```
DIAMS PC MIDAS
5 44 26 7 24 ¯11 17
```

The result has been expressed in whole percentages. Hence, *DOC* has recorded an increase in output of 5%, whereas *SLEEPY'S* output has diminished by 11%.

The function *PC* is one of the essential tools. It must also accept vectors in addition to matrices as operands.
The left-hand operand is the present value; the right-hand operand is the previous value.
It remains only for you to write it.

FIFTH THEME

A function must serve to combine two vectors into a matrix. Starting from vectors, it can serve on numerous occasions, for preparing a matrix which will be treated by another function. We will use it for the topic *"Crossed numbering"*.

```
      1 3 6 7 4 WITH 2 4 6 7 8
1 2
3 4
6 6
7 7
4 8
      'TOPIC' WITH 'SIMPLE'
SS
UI
JM
EP
TL
 E
```

SETTLEMENT OF ACCOUNTS

A company wishes to follow its accounts month by month. The names of the following headings are contained in the matrix *NECTAR*:

(*NECTAR* has 14 lines and 20 columns)

SALARIES/CHARGES
HEAVY PLANT
SMALL EQUIPMENT
SEEDS/FERTILIZER
RATES
PETROL
COMMERCIAL COSTS
DUTIES AND TAXES

8 cost items

TOTAL EXPENDITURE
CEREALS
EARLY CROPS
FRUIT

3 revenue items

TOTAL REVENUE
—— RESULT —— (Total revenue)-(total expenditure)

At the begining of the year, a matrix of 14 lines (one per item) and 12 columns (one per month), is initialized at zero. This is called *BOOK*. Then, month by month, the values actually observed for the month which has just ended are updated. For example, at the end of April, the values for the month will be entered in the fourth column of *BOOK*.

So as to avoid a task which the computer could do, we will enter only the 11 basic values, thus leaving the programme to calculate the total costs, total returns and the month's result.

The function assigned this task is called *RECEIVES*. It accepts the month number as left-hand argument, and the 11 measured values as right-hand argument as in the following example:

4 *RECEIVES* 63200 0 2856 3720 5000 1630 4169 800 11700 89650 61260

↑ for April — 8 expenditures — 3 returns

FIRST STEP

You are asked (would you believe it?) to write the function *RECEIVES*. Work on a *BOOK* which is already partly filled up to March for example.

SECOND STEP

We wish to suitably display these results for a given period. For example, for the period April-May-June, we should write:

DISPLAY 4 5 6

The result must be presented thus:

	4	5	6
SALARIES/CHARGES	63200	57800	0
HEAVY PLANT	0	12600	0
SMALL EQUIPMENT	2856	1908	0
SEEDS/FERTILIZER	3720	4212	0
RENTS	5000	5000	0
PETROL	1630	1120	0
COMMERCIAL COSTS	4169	930	0
DUTIES AND TAXES	800	2100	0
TOTAL EXPENDITURE	81357	85670	0
CEREALS	11700	9980	0
EARLY CROPS	89650	97500	0
FRUIT	61260	74600	0
TOTAL REVENUE	162610	182080	0
---RESULT---	81235	96410	0

In this example, the values for June have not yet been entered.

THIRD STEP

We do not want the function to print the months for which all the values are nil. Hence, in the above example, June should not have appeared.

Could we also print the months clearly (*JANU FEBR* ...) rather than their number?
Of course, and when assuming the function we will still use the numbers.

FIVE COLUMNS INTO ONE

The total sales of several hundred articles at 5 selling points has been recorded in the numeric matrix *LISBASKET*. These products are classified into 20 families (frozen, early crops, fish, spices, ...), a family code between 1 and 20 being attributed to each article.

The vector *LISFACO* contains the family codes associated with the articles referenced in *LISBASKET*. We assume also that *LISBASKET* and *LISFACO* are classified in increasing order of family codes.

Besides these data, two matrices of characters are also supplied:

- *LISSP*: matrix of the names of the 5 selling points (5 lines, 10 columns):

  LEEDS
  DOVER
  CARDIFF
  PERTH
  POOLE

  The matrix contains several blank columns to the right.

- *LISFA*: matrix of family names (20 lines, 15 columns):

  FROZEN
  EARLY CROPS
  FISH
  SPICES
  MEAT
  PORK PRODUCTS
  DRINKS

  ... etc ...

We intend simply to print the matrix *LISBASKET* by dividing it family by family, with an indication of the total sales for each family.

Since printing is undertaken on continuous computer paper, we want to print as many families as possible on one page, taking care however, that one family never overlaps on to two consecutive pages.

The final presentation is prescribed; it must conform with the presentation of the following example.

EXAMPLE OF PAGE SETTING

LEEDS	*DOVER*	*CARDIFF*	*PERTH*	*POOLE*	*FROZEN*
85	54	49	88	15	
73	74	78	21	12	
67	82	62	71	70	
92	83	30	36	39	
---	---	---	---	---	
317	293	219	216	136	

LEEDS	*DOVER*	*CARDIFF*	*PERTH*	*POOLE*	*EARLY CROPS*
53	59	80	44	80	
33	45	55	49	34	
25	23	57	76	13	
54	51	89	72	56	
84	62	80	24	28	
44	13	69	87	30	
---	---	---	---	---	
293	253	430	352	241	

LEEDS	*DOVER*	*CARDIFF*	*PERTH*	*POOLE*	*FISH*
25	37	83	64	23	
66	42	42	51	23	
---	---	---	---	---	
91	79	125	115	46	

LEEDS	*DOVER*	*CARDIFF*	*PERTH*	*POOLE*	*SPICES*
17	11	9	20	13	
8	10	6	11	9	
35	24	19	23	27	
6	5	9	10	5	
---	---	---	---	---	
66	50	43	64	54	

LEEDS	*DOVER*	*CARDIFF*	*PERTH*	*POOLE*	*MEAT*
85	103	98	62	77	
44	43	37	52	60	
110	103	99	122	60	... etc ...

The prescribed presentation comprises:

- In front of each family, a header comprising:
    - a blank line
    - the list of selling points, followed by the family name,
    - a blank line.
- Lines extracted from *LISBASKET*, concerning this family,
- finally; indication of the family total, with:
    - a line of dashes,
    - the total of each selling point,
    - a blank line.

In other words, for a family of *n* articles, the computer must print *n*+6 lines.

PAGE SETTING

Since the number of lines which can be printed on one page (or rather, say, a draft -screen if you want to be taken seriously by data processors!) is subject to variations according to the paper used, we will register it in the global variable *DRAFT*.

The most common paper has 66 lines per draft -screen, assumed that the draft-screen had only 33 lines, so as to clearly show the page jumps.
Moreover, the graduations have been printed on the left, so as to facilitate referencing of the lines. Obviously these graduations should not be reproduced.

This example demonstrates that it has been possible to print three families on the first draft -screen, but that there was not enough space left to print the fourth. The computer has therefore passed a sufficient number of blank lines so as to start at the top of the following draft-screen.

IMPORTANT: It is assumed that no family comprises more lines than can be printed on one draft -screen, including header and totals.

SELECTED INVOICES

We have information regarding the settling of invoices issued by a company. This information is contained in two variables:

Vector *AMOUNTS*	matrix *DATES*			
7700	7	1	7	2
12350	7	1	21	1
6300	7	1	14	3
7620	12	1	2	2
9700	12	1	28	3
33200	12	1	22	1
9100	15	2	30	4
5120	16	2	8	3
etc ...	... etc ...			

AMOUNTS is the vector of the amounts of the invoices, whereas the four columns of *DATES* represent respectively the day and month of issue of the invoices, and the day and month of their payment.

The aim is to print certain selected parts of this data.

FIRST STEP

We are asked to write a function called *PRINV*, which will have no arguments (it will work directly on the global variables *AMOUNTS* and *DATES*). It must ask which month we want to select, extract the invoices issued during this month and then print-in this order:

- the month when issued
- the day issued
- the amount
- the day and month of payment.

An example is given below. The user has requested print-out of the February invoices.

```
      PRINV
MONTH SELECTED
⎕:
      2

MONTH   DAY   AMOUNT   PAYMENT

  2      15     9100    30   4
  2      16     5120     8   3
  2      16    10600     9   3
  2      20    17430    11   5

        ... etc ...
```

SECOND STEP

We now want the month of issue to be printed clearly, with four characters only, so as to obtain the following presentation:

```
MONTH   DAY   AMOUNT   PAYMENT

 FEBR    15     9100    30   4
 FEBR    16     5120     8   3
 FEBR    16    10600     9   3
 FEBR    20    17430    11   5      ... etc ...
```

THIRD STEP

We want to improve selection in such a way as to be able to extract several months, the list of which will be entered by means of the question *"MONTHS SELECTED"*.
This list can be entered in an explicit manner, for example by answering 3 6 7 8, but we also want to constitute it by means of three auxiliary words.

These three words (variables or functions) will be:

- *ALL* which will cause all the months to be printed,
- *TO* which will serve to express a series of months in the form 6 *TO* 10 (in other words: 6 to 10),
- *EXCEPT* which will allow exceptions to be specified. For example:

  ALL EXCEPT 7 8 or again (5 *TO* 12) *EXCEPT* 8

FOURTH STEP

We wish also to carry out selections on the amounts of the invoices, by means of a second question entering into the performance of *PRINV*.

Here again, auxiliary "Tools" should provide the following types of answer:

- *ALL* — to obtain all invoices issued during the selected months,
- *AMOUNTS* ≥ 9000 — and all similar types, to carry out selection by simple comparison,
- *AMOUNTS BETWEEN* 5000 10000 — to obtain all the invoices issued during the given months, to amounts between 5000 and 10000 francs.

Note that the significance of the word *ALL* is not the same in the first and second questions.

We will encounter an example of such a selection in the fifth step.

LAST STEP

In order to improve presentation, we want the name of the month issue to appear once only, instead of being repeated as many times as there are invoices printed. An example is given below:
It is advisable to check the answer even when only one month is selected.

```
      PRINV
MONTH SELECTED
⎕:
      ALL EXCEPT 6 TO 9
AMOUNTS SELECTED
⎕:
      AMOUNTS BETWEEN 5000 8000

MONTH   DAY   AMOUNTS   PAYMENT

JANU      7     7700      7   2
          7     6300     14   3
         12     7620      2   2
FEBR     16     5120      8   3
         21     6000     25   2
         21     6300      5   6

              ... etc ...
```

NOW, THROW AWAY MY BOOK

It is not possible to introduce a large vector into the terminal in one go. Also, to introduce a large text, some must be catenated end to end with lines entered successively on the keyboard, so as to form a large vector. A function will provide for this.

In order to indicate end of text, the user will type a carriage-return.

Would you know how to write this function? Here is an example:

```
                PROSE ← INTROTEXT
          TYPE YOUR TEXT; FINISH WITH A CARRIAGE-RETURN:

          HERE IS THE FIRST LINE OF A TEXT TOTALLY DEVOID OF
          LITERARY INTEREST, BUT WHICH HAS THE ADVANTAGE OF
          DEMONSTRATING CERTAIN TYPOGRAPHICAL FEATURES.
          NOTICE THAT A COMMA IS ALWAYS FOLLOWED BY A BLANK,
          THE SEMI-COLON ITSELF BEING PRECEDED AND FOLLOWED
          BY A BLANK.  THE LINES INTRODUCED ARE, OF COURSE,
          OF UNEQUAL LENGTH.
```

The result is a vector called *PROSE*, which comprises 355 elements. The blanks have been inserted by the program at the end of each line, so as to prevent the last word of a line being stuck to the first of the following, as for example:

DEVOIDOFLITERARY

To demonstrate, here is the print-out of a part of the text obtained:

```
                200 ↑ PROSE
          HERE IS THE FIRST LINE OF A TEXT TOTALLY DEVOID OF LITR
                ERARY INTEREST, BUT WHICH HAS THE ADVANTAGE OF DEMONSTRAT
                ING CERTAIN TYPOGRAPHICAL FEATURES.  YOU W
                ILL NOTICE THAT A COMMA I
```

Of course, this text is cut by the computer "blind" when the line is complete, the above text being printed with a width of 60 characters per page ($\square PW \leftarrow 60$).

You will no doubt have guessed what the next step of this topic will be.

SECOND STEP

Since the computer does not offer good presentation, let's help it.

A text has been entered in the form of a vector and we intend to print it in a document whose width will be given as argument.
We must therefore write a function instructed to divide the text into words, and to assemble the lines obtained into a matrix having exactly the prescribed width.

As a first step, we will assume that we can cut the text at any place where a blank appears.

Here is an example of how this is done. We have requested a matrix with a width of 65 characters:

```
      65 PRINTEXT PROSE
HERE IS THE FIRST LINE OF A TEXT TOTALLY DEVOID OF LITERARY
INTEREST, BUT WHICH HAS THE ADVANTAGE OF DEMONSTRATING
CERTAIN TYPOGRAPHICAL FEATURES. YOU WILL NOTICE THAT A
COMMA IS ALWAYS FOLLOWED BY A BLANK : THE SEMI-COLON ITSELF
BEING, PRECEDED AND FOLLOWED BY A BLANK. THE LINES INTRODUCED
ARE, OF COURSE, OF UNEQUAL LENGTH.
```

It will be appropriate now to refine this rough draft. In fact, some punctuation marks (semi-colon, question or exclamation mark, colons) are preceded by a blank (there are others also). It would be unsightly to cut a text on this blank, because the following line would start with the punctuation mark. You can test your solution on this extract from the *Fruits of the earth* André GIDE:

```
      60 PRINTEXT GIDE
And now, Nathaniel, throw away my book. Shake yourself free
of it. Leave me. Leave me; now you are in my way; you hamper
me; I have exaggerated my love for you and it occupies  me
too  much. I am tired of pretending  I  can  educate anyone.
```

It can clearly be seen here that cutting the second line was badly done.
Can you do better?

"HOPE" AND "TRUTH"?

The array *HOPE* contains the sales estimates of three companies for the coming 12 months.

150	200	240	260	290	300	300	300	280	240	220	200
320	400	420	400	400	390	380	370	360	350	350	350
290	350	300	250	200	150	150	200	270	340	420	500

The array *TRUTH* contains the scales actually recorded up to a date. It is therefore an array which itself also has three lines and twelve columns, of which only the left side is fitted, until the results of the last month are known.

150	178	233	270	0	0	0	0	0	0	0	0
350	437	477	281	0	0	0	0	0	0	0	0
280	478	310	260	0	0	0	0	0	0	0	0

We wish to present a comparison between estimates and actual achievements, in the form of a single array. We will show the significant differences by the signs + or −, in accordance with the pattern below:

TRUTH COMPARE HOPE

150	150	200	178−	240	233	260	270
320	350	400	437	420	477+	400	281−
290	280	350	478+	300	310	250	260

This example shows alternate columns of estimates and actual achievements. On the right of these, the + and − signs show relative differences greater than plus or minus 10%.

For example, 178 is 11% below the 200 estimated, hence the − sign

477 is 13.6% above the 420 estimated, hence the + sign

The program must of course automatically eliminate months whose results are not yet known, so as to display only the others.

Writing the function *COMPARE* can be divided into three phases, as shown on the following page.

1/ Extraction of columns for which results are known.

2/ Imbrication of the two arrays thus obtained, by alternating their columns.
This can be obtained by means of an auxiliary function which can serve on several other occasions, since the problem raised here is very common.

Here is how such a function should be used.

If *U* and *V* are the two following matrices:

```
1 9 5 5         3 0 3 3
9 0 0 0   and   6 7 8 8
1 5 0 5         8 8 9 6
```

We can write:

```
    U IMBRICATES V
1 3 9 0 5 3 5 3
9 6 0 7 0 8 0 8
1 8 5 8 0 9 5 6
```

The function should preferably also accept vectors as operands:

```
    21 37 89 63 IMBRICATES 14 58 66 42
21 14 37 58 89 66 42
```

3/ Formatting and indicating differences by appropriate signs.

Best of luck!

EVERYTHING IS IN THE PRESENTATION

Suitably presenting an array of results is such a common problem that one should not hesitate to devote a few functions to it, which are simple but adequate for solving most typical cases.

Here for example is an array called *ARR*:

INDEX	100	150	200	250	300	350	*TOTAL*
MEN	403375	402480	399950	359667	187752	83673	1836897
	125	177	95	63	24	9	
	3227	3440	4210	5709	7823	9297	
WOMEN	695727	704573	429704	460161	84007	53334	2427506
	207	167	88	81	11	6	
	3361	4219	4883	5681	7637	8889	
TOTAL	1099102	1107053	829654	819828	271759	137007	4264403

It is scarcely pleasent to read such an array. Its presentation can be considerably improved by marking out the lines and columns at regular intervals, by means of a function.

This function will receive as arguments, besides the name of the array:

- the number of lines to be marked out,
- the position of the first (i.e. the index of the line after which it must be marked out),
- the spacing between two successive lines,
- the same three pieces of information for the columns.

The resulting array will be framed automatically, to give the array on the following page.

It is seen that the presentation obtained is more than adequate.

```
3 1 3 4 8 18 PRINT ARR
```

INDEX	100	150	200	250	300	350	TOTAL
MEN	403375 125 3227	402480 117 3440	399950 95 4210	359667 63 5709	187752 24 7823	83673 9 9297	1836897
WOMEN	695727 207 3361	704573 167 4219	429704 88 4883	460161 81 5681	84007 11 7637	53334 6 8889	2427506
TOTAL	1099102	1107053	829654	819828	271759	137007	4264403

The left-hand argument indicates that 3 lines are marked out. The first will be placed after the 1st line of data, and they will be separated from each other by 3 lines of data.

Two differences between marking out of the lines and of the columns may be observed:

1 - In order to mark out the horizontal lines, the lines of data must be SEPARATED. This is not necessary for the vertical lines, which are situated in the blank columns of the initial array.

2 - The array is formed above and below by two horizontal lines. On the right and left, it is advisable to separate the data from the framing lines by an additional blank column.

MONTE-CARLO METHOD

A factory wishes to follow closely the manufacturing costs of six important products. They therefore proceed to cost the prices of the elements which are used in their manufacture: raw materials, prefabricated components, energy, labour, etc .... which we shall henceforth call the basic prices. These basic prices are introduced in a vector, called *COSTING*, of 40 elements.

These basic prices are thus balanced differently for each of the six products. The matrix *BEARINGS* (40 lines, 6 columns) containing the balancing coefficients to be applied to each basic price for each product.

APERITIF

Calculate the six manufacturing costs.

MAIN COURSE

We wish to study the sensitivity of these manufacturing costs to variations in basic prices, so that they can be forecast during the coming months. For this, we will proceed as follows.

For each basic price, we assume a minimum and maximum value, between which it will probably fall. We will call this array *ASS*, which will be constituted manually (2 by 40).

However, these prices will not all simultaneously take their highest or lowest value. A good way of simulating the course of things consists of generating random prices between the limits given by *ASS*, and to calculate the resulting manufacturing costs.

Proceeding with *n* random selections, we will obtain *n* sets of 6 prime costs.

After these *n* tests, the six minimum manufacturing costs, and the six maximum manufacturing prices. It is interesting to compare them with the prices which we would obtain by taking only the low assumptions, or only the high assumptions.

A specimen procedure is set out on the following page.

SPECIMEN PROCEDURE

For the requirements of this example we are limited to 3 products, with only 15 elements entering into their composition. The matrix *BEARINGS* thus has 15 lines and 3 columns.

Here are the various data which enter into the process:

PROCESS:	45	61	24	34	53	36	22	24	29	17	37	61	62	11	62
○ *BEARINGS*	8	5	1	7	20	28	0	4	1	5	21	11	2	15	0
	1	20	4	0	4	15	13	5	3	2	1	0	4	5	25
	2	1	8	16	5	11	11	,8	4	16	4	5	9	5	7
ASS:	44	65	20	35	53	36	20	30	24	17	35	58	60	11	55
	46	70	30	36	60	40	25	35	35	19	38	70	70	[illegible]	62

The actual production costs amount to 4942 4530 and 3870, after computing.

We then write a function called *TESTASS*, to which we give the number of random selections required as left-hand argument and the range of price assumptions as right-hand argument.
Here is the result obtained for 10 random selections:

```
      10 TESTASS ASS
5049  4615  3993     ← minimum costs
5341  4790  4101     ← maximum costs
```

We can compare these results with those which would be obtained by applying the assumed low values, or the assumed high values, to the basic products:

- with all the lowest prices, we have: 4897 4397 3772
- with all the highest prices, we have: 5493 4982 4348

It is seen that, on comparing these extreme values *TESTASS* gives more reasonable results, certainly closer to reality.

This exercise illustrates a range of methods consisting of simulating the possible behaviour of a pattern by means of random selections. These methods are known under the generic term of "Monte-Carlo Methods". APL renders their use particularly easy, but interpretation remains the task of statisticians.

THE MOST USEFUL FUNCTIONS

It is very agreeable to have a function which prints all the functions of a workspace. We propose writing such a function here.

A loop will survey all the functions of the workspace, whose names are given by ⎕*NL* 3. Each function will thus be printed by means of its canonical representation ⎕*CR*. For this subject see the course book pages 270 and 273.

FIRST DRAFT

As a first step, we will print all the functions in alphabetical order, following the presentation of the following page as closely as possible.

We will simplify the problem by assuming that the names of the functions of the workspace are made up from only the 26 letters of the alphabet (not underlined) and the 10 numbers. We will also assume that they are short enough to enable the classifying method shown on pages 174-175 to be applied.

IMPROVEMENTS

As a second stage, we will assume that the workspace contains locked functions and functions which contain no instructions, which may occur in error. Since these empty and locked functions cannot be printed their names will be put on one side and printed at the end of the work.

Moreover, it would be absurd for the *LIST* function which prints the others to be itself printed. Hence, it is advisable to eliminate it from the list of functions to be printed, without locking it.

You will recall that the canonical representation of a locked function is a matrix of dimensions 0 0.

SPECIMEN OF PRINT-OUT

```
        LIST

--------------------------------------------------------------- CONTRACT

     R←CONTRACTER M;DIM
[ 1] DIM←ρM
[ 2] R←DIM,(M≠0)/M,[0.5]ιρM←,M

--------------------------------------------------------------- DERIVE

     R←DERIVE POL
[ 1] POL←((ρPOL)-1) TAKE POL
[ 2] R←POL×(1+ρPOL)-ιρPOL

--------------------------------------------------------------- MOB

     M←N MOB X
[ 1] M←0,+ X
[ 2] M←(N↓M)-(-N)↓M
[ 3] M←M÷N

--------------------------------------------------------------- AREA

     S←SIDES AREA;P
[ 1] P← 0.5 × +/SIDES
[ 2] S←(×/P,P-SIDES)* 0.5

--------------------------------------------------------------- TITLE

     R←A TITLE B;BLANKS;WIDTH
[ 1] WIDTH←(ρB)[2]
[ 2] BLANKS←⌊0.5×WIDTH-ρA
[ 3] A←WIDTH↑(BLANKSρ' '),A
[ 4] R←A,[1]'-',[1]B

--------------------------------------------- EMPTY AND LOCKED FUNCTIONS

DIVI
PC
EXCEPT
WSDOC

--------------------------------------------- PRINT-OUT FINISHED
```

SEARCHING FOR SKILLED WELDERS

Some trades are subjected to very strict controls. As an example we will take welders, who can perform only certain special jobs after obtaining the necessary qualification: welding of high pressure piping, inert-gas welding, welding on to very thick materials, etc ...

Some skills are maintained permanently, others however, are lost if the welder has not had occasion to practice them for some time on the worksite. Each welder is therefore studied, and the date of the last occasion on which he has practiced his skills is carefully noted.

In order to simplify the exercise, we have assumed that these dates comprised only the years: 78, 79, etc ...
The qualifications are designated by a numeric code of three figures, and we assume that a welder has a maximum of five qualifications

DATA

Here is the information which has been collected for 10 welders:

COOPER	308	833	0	0	0	75	81	0	0	0
FLANAGAN	312	313	512	833	0	80	79	82	81	0
JONES	409	641	642	0	0	80	80	78	0	0
JOYCE	312	642	685	0	0	77	79	81	0	0
LEEVES	685	831	409	312	833	82	76	79	79	81
LITTLEJOHN	308	409	0	0	0	72	81	0	0	0
MATTHEWS	409	512	642	685	0	82	81	78	81	0
NEVILLE	312	500	641	685	409	78	82	78	81	81
ROY	512	0	0	0	0	81	0	0	0	0
TIPLER	500	833	0	0	0	82	81	0	0	0

i.e, from left to right:

- *WELDERS* : matrix of the welders' names (10 lines, 10 columns)
- *WELQUAL* : matrix of the qualifications obtained
- *WELDATE* : matrix of the dates. It indicates in which year each skill was practiced for the last time.

As all the welders have not obtained the same number of qualifications, the lines of the matrices *WELQUAL* and *WELDATE* are completed with zeros.

We also have the two following vectors:

QUALIF	*PER*
308	0
312	5
313	1
409	2
500	1
510	1
512	1
641	5
642	0
675	3
681	1
685	2
831	5
833	1

QUALIF is the vector of all the qualifications possible, whereas *PER* indicates, for each qualification, its validity period, in years. The zeros indicate the qualifications which are acquired permanently.

Comparison of all this information enables certain facts to be established. We will study the case of the welder *FLANAGAN* for example.

He acquired (or renewed) qualifications 312 313 512 and 833 in 1980 1979 1982 1981 respectively.
Qualification 312 being valid for 5 years, he will have to renew it in 1985 at latest. Qualification 313 is valid for only one year, he should have renewed it, and it is therefore lost, etc ...

THREE IMPORTANT QUESTIONS

You are asked to write three different functions, each of which must answer a different requirement of the company chief.

FIRST QUESTION:

For a particular worksite we are looking for welders possessing a given qualification. We require a list of all the welders possessing this qualification. We can generalise in the case where we require welders possessing one or other of the qualifications featuring in a given list.

For example, we will look for welders who have either qualification 308, or qualification 312:

```
WELD1 308 312
```

COOPER
FLANAGAN
JOYCE
LEEVES
LITTLEJOHN
NEVILLE

SECOND QUESTION:

The controls imposed on certain qualifications have been modified on a given date. We are looking for welders who have renewed this qualification after this date.

For example, the ruling relating to qualification 409 was modified in 1980. We will obtain the welders who have renewed this qualification since this date as follows:

```
409 WELD2 80
```

LITTLEJOHN	81	the function prints the name and date of renewal.
MATTHEWS	82	
NEVILLE	81	

On the other hand, the welders *JONES* and *LEEVES* do not appear in this list, because they renewed their qualification 409 in 1980 and 1979 respectively.

THIRD QUESTION:

Assume we are in 1982. We are looking for welders for whom one or more qualifications will expire this year, or who have already lost one or other qualification, through not renewing it in time.

For each welder in this situation, the program will print the qualification or qualifications to be renewed. Those which must normally be renewed this year are printed normally. Those whose renewal date has passed are lightly underlined.

An example of this is given on the following page. The year is given as operand, so as to enable estimation studies to be undertaken, which the use of ⎕*TS* would not provide.

WELD3	**82**		
COOPER	833		
FLANAGAN	313	833	
JONES	409		
JOYCE	312		
LEEVES	831	409	833
MATTHEWS	512		
ROY	512		
TIPLER	833		

Taking the case of the welder *FLANAGAN*, he will have to renew his qualifications 312 and 512 in 1982 and 1983 respectively. On the other hand, it appears from the above that he will have to renew his qualification 833 in 82, and that he has already lost his qualification 313.

THE PLOT THICKENS

Obtaining certain qualifications automatically confers qualifications of a lower order on their ladder. The matrix *EQUIV* gives the list of these equivalences:

QUALIF	*EQUIV*		
308	0	0	0
312	0	0	0
313	308	0	0
409	0	0	0
500	308	0	0
510	308	312	0
512	308	409	313
641	0	0	0
642	0	0	0
675	0	0	0
681	641	642	0
685	0	0	0
831	0	0	0
833	831	0	0

Due to these equivalences, the welder *ROY*, who has qualifications 512, also has qualifications 308, 409 and 313.

You are required to adapt the functions *WELD*1 and *WELD*2 to take this new information into account. We will not attempt to modify function *WELD*3.

For example:

```
      WELD1B 308
COOPER
FLANAGAN                 equivalence 512 → 308
LITTLEJOHN
MATTHEWS                 equivalence 512 → 308
NEVILLE                  equivalence 500 → 308
ROY                      equivalence 512 → 308
TIPLER                   equivalence 500 → 308
```

FOR AESTHETICS ONLY

The function *WELD*1 displayed only the name of welders who have a given qualification. It would be eloquent to display the list of his qualifications and their renewal dates for each.

```
      WELD1C 409
JONES        409 80   641 80   642 78
LEEVES       685 82   831 79   409 79   312 79   833 81
LITTLEJOHN   308 72   409 81
MATTHEWS     409 82   512 81   642 78   685 81
NEVILLE      312 78   500 82   641 78   685 81   409 81
```

In this presentation, we have not taken equivalences into account. Each qualification is immediately followed by its renewal date and the zeros have disappeared.

Imbrication of the columns of two arrays has already been the subject of the topic *"Hope" and "Truth"*. Cancelling of the zeros is easy to achieve if we print the lines one by one by means of a loop. It is more advantageous however, to find a solution without a loop, which will be useful in many other causes.

This is slightly more complicated, but much more exciting!

WILL IT BE FINE TOMORROW?

SALES is an array in which we have recorded the sales of articles in a shop during the last six months. The matrix *OBJECTS* contains the names of the articles in the catalogue:

OBJECTS	*SALES*					
SUNTAN LOTION	3	12	65	262	581	488
BATHING MAT	0	1	0	5	60	32
RUBBER RING	1	2	2	1	5	9
SUNSHADE	7	2	0	9	18	24
UMBRELLA	127	133	60	11	3	3
RAINCOAT	52	36	32	41	27	30
SNOW-BOOTS	2	6	8	7	4	2

Based on *SALES*, we wish to establish a starting point for estimates for the next six months. These estimates will be improved later, and we will merely place the next six months in relation to the values for the past six months.

A function will display the articles one after the other and the department head will indicate the total sales which he forcasts for the coming six months.

The function will have to be written in such a way that he can answer in one of the following five ways, for each product:

- he types 6 different values for each month,
- he can type a single value; the computer will have to repeat it for the coming six months,
- if he types *I*, the values for the last six months will be replayed *I*dentically,
- if he types *L*, the *L*ast value known will be repeated 6 times,
- if he types *A*, the *A*verage of the last six months will be repeated.

Since the aim is not to perform a very delicate operation, we will assume that the user does not commit any handling error, and follows this method of working without error.

You are offered two solutions for solving this project.

FIRST SOLUTION

We introduce the values by means of a simple QUAD (⎕). The introduction of numeric values therefore poses no problem, but the use of the letters *I*, *L* and *A* assumes that we have defined the variables or functions called *I*, *L* and *A*.

For example, here is how the use of such a program would be presented. The result is placed in the variable *FUTURE*, for eventual use:

```
      FUTURE ← OBJECT EXTRAPOLATE SALES
SUNTAN LOTION
⎕:
      500 200 100 20 20 100        we introduce 6 different values
BATHING MAT
⎕:
      A                            we are going to use the average of
RUBBER RING                        the last six months
⎕:
      5                            5 per month for 6 months
SUNSHADE
⎕:
      I                            Same value as the last six months
UMBRELLA
⎕:
      3 3 5 10 20 30
RAINCOAT
⎕:
      L                            6 times the last sale
SNOW-BOOTS
      A
```

The results obtained would be as follows:

FUTURE

500	200	100	20	20	100
16	16	16	16	16	16
5	5	5	5	5	5
7	2	0	9	18	24
3	3	5	10	20	30
30	30	30	30	30	30
4	4	4	4	4	4

The averages have been rounded off to the nearest lower whole number.

SECOND SOLUTION

We introduce the values by means of a QUOTE-QUAD (⍞). We must therefore detect the presence of the letters *I*, *L* or *A* in the typed text, and perform the corresponding work.
On the other hand, to accept the introduction of numeric values, we must know the EXECUTE function (⍎).

Here is how we would perform the same work as beforehand with a function written in this way.

```
          FUTURE ← OBJECTS EXTRAPOLATE SALES
SUNTAN LOTION     500  200  100  20  20  100
BATHING MAT       A
RUBBER RING       6
SUNSHADE          I
UMBRELLA          2  3  5  10  20  30
RAINCOAT          7
SNOW-BOOTS        A
```

The results would be identical.

Here we have used what is called the "bare-output" technique, which enables the computer to display a text, and to await the answer on the same line (see course book, page 107).
Presentation is distinctly more agreeable than the preceding example, if there is ever an error in entering the data.

If your knowledge is sufficient, write both solutions (⎕ and ⍞) and compare them. This is particularly instructive.

CROSSED NUMBERING

In order to follow the results of a commercial group, we have constituted three vectors of the same length, having one element per signed contract:

1 - The vector *AMOUNTS* contains the amounts of all the contracts signed.

2 - For each contract we find the registration number of the commercial engineer who undertook the business in the vector *COMM*.

3 - A binary vector, *ONEW*, specifies for each contract if it was concluded with an old client (0), or with a new client (1).

For example:

AMOUNTS :	7700	12350	6300	7620	9700	33200	9100	5120	....
COMM :	420	420	471	452	420	519	420	953	....
ONEW :	1	1	1	0	0	1	0	0	....

FIRST STEP

We wish to draw up an array giving the number of old or new clients who have signed a contract with each commercial engineer, according to the model below.
The array is bordered to the right and below by the totals of the lines and columns.

CROSS ONEW WITH COMM

old	23	11	2	13	8	6	63
new	10	3	3	8	7	3	34
totals	33	14	5	21	15	9	97
	420	*452*	*471*	*519*	*833*	*953*	*Total*

The references have been typed a posteriori in italics to facilitate interpretation of the result.

NOTES

1 - The function *WITH* serves only to incorporate the two operands (*ONEW* and *COMM*) into one matrix, in such a way that *CROSS* receives a single operand. The form of the phrase thus constituted conforms move to normal usage.

2 - In this case, we know in advance the different values which the operands can take: 0 and 1 for *ONEW*, a list of known registration numbers for *COMM*.

However the solution adopted must be more general, so as to accept any operands where possible values are not necessarily indexed in advance. Such generalisation is useful if *ONEW* can take, for example, 4 different values instead of 2:

0 = old clients,

1 = new clients,

2 = administrations and public services,

3 = internal lending between subsidiaries of the same individual group.

SECOND STEP

We wish to improve presentation by automatically adding the necessary references, which had been typed a posteriori in the first step.

The width of the columns of numbers (format) will be defined and controlled by the variable *FOR*, global to the workspace. In the example below, *FOR* equals 8.

```
          CROSS ONEW WITH COMM
            420     452     471     519     833     953  TOTALS
      0      23      11       2      13       8       6      63
      1      10       3       3       8       7       3      34
 TOTALS      33      14       5      21      15       9      97
```

THIRD STEP

We wish now to balance these results by the amount of the contracts, so as to find turn-over achieved by each commercial engineer for each category of clients.
As in the first step, we will make do with rather crude presentation, without headings:

```
      AMOUNTS BALANCE ONEW WITH COMM
 399450  254550   46300  254750  105750  146700  1207500
 125400   27700   58650  213650  105950   51400   582750
 524850  282250  104950  468400  211700  198100  1790250
```

Here again the results array is bordered by its totals.

LAST STEP

By combining the information calculated in the preceding steps, it is possible to make the following appear, for each case indexed:

- turn-over achieved,
- number of clients,
- average turn-over per client.

And to our satisfaction, we will create a fine array, with headings, as follows:

AMOUNTS BALANCE ANEW WITH COMM

	420	452	471	519	833	953	*TOTALS*
0	399450	254550	46300	254750	105750	146700	1207500
	23	11	2	13	8	6	63
	17367	23141	23150	19596	13219	24450	19167
1	125400	27700	58650	213650	105950	51400	582750
	10	3	3	8	7	3	34
	12540	9233	19550	26706	15136	17133	17140
TOTALS	524850	282250	104950	468400	211700	198100	1790250
	33	14	5	21	15	9	97
	15905	20161	20990	22305	14113	22011	18456

In the above restitution, *FOR* had been fixed at 9.

Carefully keep the functions written for this subject. They are very general, and can be of assistance in many everyday situations

If aestheticsare an important factor to you in the pursuance of data processing with terminals users, it is always possible for you to include, in the function *BALANCE* the function *PRINT*, written in the topic "*Everything is in the presentation*".
The result is certainly worthwhile.

A DIFFICULT CHOICE

Some manufactured products, numbering about 500, are referenced by a product code comprising a letter followed by three figures:

```
A529
P402
P747    etc ...
```

The letter represents the range, whereas the numeric part represents the article number in the range. Consequently, two products can have the same numeric part and belong to different ranges, for example: E410 and Y410.

There are several ways of representing the list of these codes. According to the representation adopted, seeking a code will be more or less complicated, longer or shorter.

We will later suggest several representations. For each of these structures it will be up to you to write a function for searching a given code in the list of all the codes.

If the product exists, the function must repeat its index on the list:

```
      FIND 'E410'
127
```

If it does not exist, the function must give the result zero:

```
      FIND 'U239'
0
```

It is especially advisable to test seeking codes which have the same numeric part but which belong to different ranges.

FIRST STRUCTURE

A vector called *PRORAN* will contain the representative letter of the range of 500 products. A second vector called *PRONUM* will contain the numeric part of the codes in numeric form.

This structure is probably rather complex to manage, but it is useful if we wish to make selections on the range code.

SECOND STRUCTURE

We form a matrix of characters containing one code per line by means of the following transformation:

PROMAT ← *PRORAN*, 3 0 *VERT PRONUM*

Looking for a code thus amounts to looking for a character string in a matrix of characters. This is much easier.

THIRD STRUCTURE

We combine the letter and the numbers into a single numeric value by means of decoding, as we would for alphabetical classification (see course book, page 174)
Since the codes are composed of the 26 letters and 10 numbers, decoding must be carried out in base 36 (or in a higher base, for example, base 100).

ALPHANUM ← '0123456789*ABCDEFGHIJKLMNOPQRSTUVWXYZ*'

PROVEC ← 36 ⊥ *ALPHANUM* ⍳ ⍉ *PROMAT*

Each line of *PROMAT* gives rise to a number. *PROVEC* is thus a numeric vector of 500 values.

The user himself will continue to designate products in the same way:

FIND '*K*745'

Subjecting the code sought to the same transformation, leads to seeking a numeric value in a numeric vector.

FOURTH STRUCTURE

With the preceding structure, setting up and seeking errors are complicated by the fact that there is no connection between the external form of a code (E403) and its representation in *PROVEC* (here 706360).

A much simpler alternative consists of keeping the numeric part unchanged and replacing the letter by its position in the normal alphabet:

E403 becomes 5403
Z106 becomes 26106

Here again the external form of the codes remains unchanged for the user.

FIFTH STEP

We can intuitively estimate which presentation allows the quickest searching, but it is interesting to compare such intuition with the precise measured results.

For this purpose write a function which calculates the computing time required for execution of an expression *N* subsequent times. This calculation will be achieved by the difference between ⎕*AI*[2] before and after execution.

Example:

```
        CALTIME 10
EXPRESSION: FIND1  'E562'
TIME =  530

EXPRESSION: FIND2  'E562'
TIME =  436

EXPRESSION: etc...
```

The function's argument indicates how many times each expression must be executed, so as to limit measurement errors.

Then the function asks the user to introduce the expressions. We called the functions corresponding to the structures 1,2, etc..., *FIND*1, *FIND*2 etc...

The function *CALTIME* thus executes the expressions introduced, and indicates the computing time used. Of course, the times indicated above are highly unrealistic. We will leave you to undertake your own tests.

LAST SUGGESTION

Instead of representing the range by a letter, we could use entirely numeric product codes, as with *PROVEC*. For curiosity, calculate the time gained which would result from the user using such codes:

*FIND*6 12702 (numeric argument)

THE CARROT AND THE STICK

A numeric array of L lines and C columns represents C successive values of L bits of information.

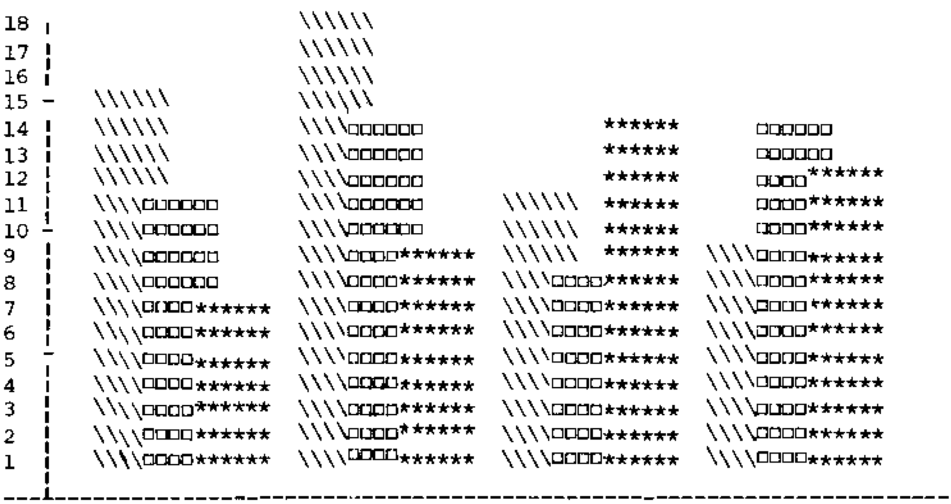
For example, the array *PROCAR* is a 3×4 matrix which represents, for 4 consecutive years, the carrot production of three countries.

PROCAR

15	18	11	9
11	14	8	14
7	9	14	12

We wish to compare these values by representing them by sticks of different "colours", in accordance with the model below. Instead of colours, we have used the following graphic symbols:

\\\\\\\\	for the first country
□□□□	for the second country
****	for the third country



In order to maintain clear legibility of the diagram, we will observe the following presentation:

- the vertical scale is equal to 1,
- each stick is 6 characters wide,
- a stick encroaches on the preceding stick by 2 characters,
- each group of sticks is separated from the preceding one by 3 spaces.

Of course, the matrix used as an argument must be able to be any dimensions.

In this exercise, sticks representing the same column of the matrix are arranged side by side so as to form a group of sticks. They could also be placed one above the other, in such a way that the total height represents the total carrot production in three countries. This arangement is more complicated to obtain; it will be subject of another exercise entitled *"Spokes in the wheels"*.

IN THE TIME OF THE PYRAMIDS

All companies must keep a register of the starting and leaving dates of its employees. We have extracted a numeric matrix *ES* from this which has as many lines as employees indexed, and 4 columns representing respectively:

- the month and year recruted
- the month and year of leaving. For people still present in the company. these last two values are zero.

For example:

```
 3  70  8  71
 3  70  0   0
 4  71  5  71
10  71  9  81
... etc ...
```

The first person, recruited in March 1970, left in August 1971, whereas the second. recruited in the same month is still present in the company,

FIRST STEP

We want to establish, at a given date (month, year), the distribution of personnel present by years of service (on a scale from 0 to 20).
For example, in March 1976 the personnel were distributed thus:

```
      CUT 3  76
11 5 4 0 2 1 1 2 0 0 1 1 1 0 0 0 0 0 0 0 0 29
```

In other words on this date 11 people had less than one year's service, 5 had one year, 4 had two years, etc ...
In total, 29 people were present on this date.

As the dates are not specified by the day, it will be advisable, for a given month:

- to count as present people recruited during this month,
- not to count people who left this month,
- to count one year of service for these people recruited the same month of the previous year, etc ...

Now write this function.

SECOND STEP

We wish to know the pattern of this length of service pyramid in the course of time, by applying "cuts" in the Decembers of several successive years, whose list will be given as an argument.

In the function below, we have given as left-hand argument the length of service scale on which we wanted to work, and as right-hand argument the years on which the study took place. In both cases, we have used the service function *TO*.

```
(1 TO 12) PYRAMIDS 74 TO 79
```

	1	2	3	4	5	6	7	8	9	10	11	12	
74	9	2	2	2	0	2	0	0	2	1	0	0	20
75	5	4	1	2	1	0	2	0	0	2	1	0	18
76	9	4	2	0	2	1	0	2	0	0	2	1	23
77	6	6	2	0	0	2	1	0	2	0	0	2	21
78	9	4	2	2	0	0	2	1	0	2	0	0	22
79	9	5	2	0	2	0	0	2	1	0	2	0	23

FLASH

We wish to present the results of the various subsidiaries of a large industrial group and calculate certain totals and percentages.

We assume that the information is already collected and the calculations made. Hence we have three data:

- the vector *CODES* which contains the codes enabling the lines of results to be designated,
- the array *POSTS* which contains the name associated with each post, of a maximum of 30 characters,
- the array *RESULTS* which contains the numeric information relating to the last three months.

CODES	*POSTS*	*RESULTS*		
10	*FRANCE*	965	214	280
12	*ITALY*	388	666	214
15	*GERMANY*	686	659	822
16	*ENGLAND*	323	529	450
50	*E.E.C.*	2362	2068	1766
18	*U.S.S.R.*	484	228	952
51	*EUROPE TOTAL*	2846	2296	2718
11	*ALGERIA*	183	246	164
13	*TUNISIA*	429	328	222
19	*MOROCCO*	805	510	415
20	*IVORY COAST*	507	828	938
52	*AFRICA TOTAL*	1924	1912	1739
60	*EUROPE MANAGEMENT*	5397	4583	4760
14	*CANADA*	524	555	640
17	*UNITED STATES*	835	780	516
23	*MEXICO*	956	669	495
61	*AMERICA MANAGEMENT*	1688	1629	1348
21	*EUROPE-AMERICA SALES*	988	959	866
22	*AMERICA-EUROPE SALES*	361	584	563
70	*OVERALL TOTAL* -------->	7085	6212	6108
80	*E.E.C. PERCENTAGE*	33	33	29
81	*EUROPE PERCENTAGE*	76	74	78
82	*AMERICA PERCENTAGE*	24	26	22

From this data we wish to print flashes of partial information, destined for various authorities in the company.

PRINCIPLE

These partial arrays will be printed by means of a process in two steps.

\- 1 - First phase

Each restitution is defined in advance by means of the function *DEFREST*, by the two following pieces of information:

- its header (30 characters maximum)
- the list of numbers of lines involved in this restitution. The intentionally placed zeros in this list indicate the places where blank lines will have to be included when printing.

Hence we can define numerous different restitutions in advance. The headers constitute the matrix *HEDREST* (*n* lines and 30 columns), whereas the codes of the lines to be printed are contained in the numeric matrix *CODREST* (same dimensions).

For example:

```
      DEFREST
ARRAY HEADER....AFRICA ZONE RESULTS
LINES TO BE PRINTED
⎕:
      50  0  11  13  19  20  0  52  60
THIS ARRAY WILL BEAR THE NUMBER 4
```

\- 2 - Second phase

A second function *PRIREST*, serves to print an array already defined, designated by its order number in the list of restitutions.

An auxiliary function, *ARRAYS*, gives the numbered list of predefined restitutions.

An example is given on the following page.

It remains only for you to write the functions *DEFREST*, *PRIREST* and *ARRAYS* in accordance with these directions ..... a real pleasure!

SPECIMEN USE

We have seen on the previous page, how to define a new restitution. Imagine that we have defined 4 restitutions in this way, the intermediary arrays taking the following values:

HEDREST
EUROPE RESULTS
OVERALL SYNTHESIS OF THE GROUP
AMERICA RESULTS
AFRICA ZONE RESULTS

CODREST

10	12	15	16	0	50	0	18	0	51	0	0	80	81	0	0	0	0	0	0	0
51	52	21	22	0	60	0	0	61	22	21	0	0	70	0	80	81	82	0	0	0
14	17	23	0	22	21	0	61	0	82	0	0	0	0	0	0	0	0	0	0	0
50	0	11	13	19	20	0	52	60	0	0	0	0	0	0	0	0	0	0	0	0

.... etc ....

We can therefore print one of these arrays, for example the second as follows:

PRIREST 2

```
OVERALL SYNTHESIS OF THE GROUP               17  1  1982
------------------------------------------------------------

51 EUROPE TOTAL                           2846  2296  2718
52 AFRICA TOTAL                           1924  1912  1739
21 EUROPE→AMERICA SALES                    988   959   866
22 AMERICA→EUROPE SALES                    361   584   563

60 EUROPE MANAGEMENT                      5397  4583  4760

61 AMERICA  MANAGEMENT                    1688  1629  1348
22 AMERICA→EUROPE SALES                    361   584   563
21 EUROPE→AMERICA SALES                    988   959   866

70 OVERALL TOTAL  ------>                 7085  6212  6108

80 E.E.C. PERCENTAGE                        33    33    29
81 EUROPE PERCENTAGE                        76    74    78
82 AMERICA PERCENTAGE                       24    26    22
```

The lines have indeed been printed in the order indicated by *CODREST*[2;], passing one blank line at each zero encountered. The array is headed by *HEDREST*[2;], supplemented by the date.

WEEK-END SAILING AT BRIGHTON

A company committee organises group outings which it partially finances. The rest is financed by voluntary donations from the participants. This donation can be raised, especially for employees who bring their family, and it can be divided into several instalments.
We will assume that the donations are identical for all those participating in the outing, whether they are employees, spouses, or members of their family.

We wish to equip ourselves with a tool for following financing of these outings. Various functions will have to perform the processings described hereafter.

INFORMATION INPUT

ORGANISING AN OUTING

When an outing is planned, we enter the following data into the computer:

- description of the outing (30 characters maximum)
- total individual donations.

Moreover, the computer automatically attributes a number to this outing, so that it is easy to designate it in other programs.

We call:

- *OUTNAMES* the matrix of outings descriptions,
- *OUTPRICES* the vector of participation prices,
- *OUTCODES* the vector of numbers attributed to the outings.

Example:

```
                PLAN
        OUTING DESCRIPTION?
        CHEDDAR GORGE - 2 DAYS
        INDIVIDUAL DONATION?
        ⎕:
                600
        --- THIS OUTING WILL BEAR CODE 8
```

INPUT OF REGISTRATIONS

Registrations are entered as far as possible outing by outing in the following way:

- outing code number. The computer displays the description in full, as a check.
- Employee's registration number,
- number of places reserved. From this information, the computer displays the amount due from the employee,
- amount of initial instalment.

Then we pass to the next employee, for the same outing.

This assumes that we have two data relating to the personnel:

- *PERNAMES* matrix of the employee's names (12 columns)
- *PERREGS* vector of the employee's registration.

FURTHER INSTALMENTS

When an employee deposits a further instalment, the computer asks for his registration and the outing number. This information enables it to display the total instalments already paid and the amount still owing.

Examples of both these processings are given on the following page.

REGISTRATIONS

Start of program

OUTING CODE
⎕:

5
ONE WEEK IN MOROCCO

We introduce the outing code and the computer displays its name.

REGISTRATION, NUMBER OF PARTICIPANTS
⎕:

122 2
DONATION: 2400
FIRST INSTALMENT
⎕:

The computer displays the sum due in terms of the number of participants.

800

Partial instalment

REGISTRATION, NUMBER OF PARTICIPANTS
⎕:

We pass on to the next person, still for the same outing.

403 1
DONATION: 1200
FIST INSTALMENT
⎕:

1200

REGISTRATION, NUMBER OF PARTICIPANTS
⎕:

END

We have finished for this outing, and we proceed to the next.

OUTING CODE
⎕:

8
CHEDDAR GORGE - 2 *DAYS*

If we had finished we would type *END* as outing code.

REGISTRATION, NUMBER OF PARTICIPANTS
⎕:

580 4
DONATION: 2400
FIST INSTALMENT
⎕:

1000

etc.........

INSTALMENT

Further instalments input program

REGISTRATION, OUTING
⎕:

122 5
ALREADY PAID: 800, *STILL OWING*: 1600
INSTALMENT
⎕:

This information must be checked.

1000

etc....

RESTITUTIONS

An initial program enables the state of the planned outings to be determined. We print for each one:

- the outing code and the complete description,
- the amount of the individual donation,
- the number of participants,
- the budget which this represents (i.e. the product of the two preceding numbers),
- the total amount of the payment already collected for this trip.

TRIPS

CODE	*OUTING*	*PRICE*	*PART*	*BUDGET*	*RECEIVED*
5	*ONE WEEK IN MOROCCO*	1200	15	18000	18000
6	*SOUTH COAST, DEVON*	750	8	6000	5500
7	*WEEK-END SAILING AT BRIGHTON*	500	12	6000	2300
8	*CHEDDAR GORGE - 2 DAYS*	600	43	25800	9600

A second program displays the outings requested by each employee, showing the following information:

- employee's registration number and name,
- codes of outings requested with, for each one:
  - number of people participating,
  - corresponding total amount,
  - total amount already paid,
  - amount still owing.

STATE

REG NO.	*NAME*	*OUT*	*PART*	*TOTAL*	*RECEIVED*	*OWING*
115	*MORAN*	6	2	1500	1000	500
		8	1	600	600	0
122	*TIPLER*	7	3	1500	1500	0
		8	3	1800	500	1300
		6	1	750	750	0
241	*BOYLE*	6	1	750	200	550
245	*EASEMAN*	7	2	1000	500	500

Finally, for each outing, we require a list of the participants. In order to save space, we will write 3 names per line, as follows.

PARTICIPANTS

SOUTH COAST, DEVON	*MORAN*	2	*BOYLE*	1	*TIPLER*	1
	JOYCE	3	*MATTHEWS*	2		
WEEK-END SAILING AT BRIGHTON	*TIPLER*	3	*EASEMAN*	2		
CHEDDAR GORGE - 2 DAYS	*MORAN*	1	*TIPLER*	3	*FORD*	1
	GILLAM	2	*BICKERSTETH*	4	*GOODEVE*	2
	COOMBER	3				

Only outings for which there are participants appear here. For each outing, we find the list of participants associated with the number of people participating in each one.

CONCLUSION OF AN OUTING

When an outing is over, a function enables it to be erased from the list of planned outings. However, this function checks that all the participants have settled their donations. If this is not the case, it will not erase it.

For example:

```
      ERASEOUT
WHICH OUTING?
⎕:
      6
UNSETTLED DONATIONS
```

The function refuses to erase.

```
      ERASEOUT
WHICH OUTING?
⎕:
      5
IT'S DONE
```

This time it is erased.

CAN YOU UPDATE?

The personnel of a company are divided into 6 hierarchial categories. We check the state of this on the first of January of two consecutive years, and we note:

- in *REG NO*1 and *CAT*1 the registrations and categories of all the employees present on the first check,
- in *REG NO*2 and *CAT*2 the same information for the employees present on the second check. Here we find nearly all the employees who featured in *REG NO*1, plus some new ones.

These pieces of information are therefore not of the same length, and moreover, they are not classed in the same order.

FIRST STEP

It is required to calculate the personnel updating matrix:

		NEW CATEGORY						
		1	2	3	4	5	6	DEPARTURES
ENGAGEMENTS		16	18	11	7	3	8	0
OLD CATEGORY	1	66	17	8	2	0	0	3
	2	0	73	27	6	5	0	7
	3	0	0	49	24	5	4	9
	4	0	0	0	46	14	8	3
	5	0	0	0	0	27	11	2
	6	0	0	0	0	0	52	4

We read from this matrix that there have been 16 engagements in category 1, 18 in category 2, 11 in category 3, etc....

We can also see that among the people who were in category 1, 66 are still there, 17 have been promoted to category 2, 8 to category 3, two lucky ones have been promoted to category 4 and 3 have left the company.

ARRAY OF MANPOWER TRANSFERS

From this information it is possible to establish transfers of personnel from one category to another.
For each category we wish to show:

- the initial manpower at the first check
- detail of entries:
  - by promotion from a lower category,
  - by direct engagement.
- the total entries (total of the two preceding lines),
- detail of departure:
  - by promotion to a higer category,
  - by finally leaving the company.
- the total departures (total of the two preceding lines),
- the final manpower, at the second check.

A simple function links both steps:

```
      TRANSFERS
UPDATING MATRIX

      16  18  11   7   3   8   0
      66  17   8   2   0   0   3
       0  73  27   6   5   0   7
       0   0  49  24   5   4   9
       0   0   0  46  14   8   3
       0   0   0   0  27  11   2
       0   0   0   0   0  52   4
```

ARRAY OF MANPOWER TRANSFERS

A column on the right contains the totals

CATEGORY	1	2	3	4	5	6	
INITIAL MANPOWER	96	118	91	71	40	56	472
PROMOTIONS	0	17	35	32	24	23	131
ENGAGEMENTS	16	18	11	7	3	8	63
TOTAL ENTRIES	16	35	46	39	27	31	194
PROMOTIONS	27	38	33	22	11	0	131
DEPARTURES	3	7	9	3	2	4	28
TOTAL OUTGOING	30	45	42	25	13	4	159
FINAL MANPOWER	82	108	95	85	54	83	507

Could you write the function *TRANSFERS*?

SPECIAL PRINTING

Information concerning a group of n people is represented by various variables:

- *NAME* is the matrix of the n names ($n$lines, 9 columns),
- *AGE* is the numeric vector of the n ages,
- *SEX* is a vector of characters, representing sex,
- *DEPT* is the numeric vector giving the company department of these n people,
- *MARSTA* is the vector which represents the marital status of the people by one of the following characters, *S*, *M*, *D* or *W*,
- *REG* is a numeric vector containing the registration no. the people,

The position of the information relating to the same individual is the same in all the variables.
We have done this in such a way that all the variable names have a maximum of six characters.

We wish to be able to print all or part of this information, for all or some of the people, by a process in three steps.

- 1 - We define a character which the print-out must elucidate by indicating the names of the variables to be visualized, in the order in which we want to see them appear. Some may appear several times as in the following example.

CHARACTER
DATA TO BE PRINTED ..... *REG No. NAME AGE SEX MARSTA REG No.*

------> *IT'S DONE*

Of course, the function *CHARACTER* must check that the names of the variables requested are correct.

- 2 - A print-out of the information is required concerning various people designated by their index in the variables.
For example, in order to print the information concerning the 3rd, 11th, 2nd, and 27th persons, we will write:

PRINT-OUT 3 11 2 27

REG No	NAME	AGE	SEX	M.S.	REG No.
645	VANNIEL	46	M	M	645
285	HURTUBISE	48	M	S	285
949	HUREL	34	M	M	949
712	MIALON	28	F	S	712

- 3 - We will be able thus to think about printing lists of people selected by certain criteria, eventually classifying them according to certain other criteria, as shown in the course book, pages 361 to 363.

For example: *PRINT PEOPLE* (*AGE*>32) *AND* (*DEP*=75)

Or: *PRINT* (*PEOPLE SEX* = '*F*') *ACCORDING TO AGE*

ADVICE

We will use two matrices *TAFOR* and *TAHED* to contain the printing formats of the various variables and the headers which must feature above each column of the first array respectively. Both these matrices will be contributed manually.

The first function, *CHARACTER*, will elaborate two global variables which will be used by *PRINT-OUT*:

- *ZONES* : List of the numbers of the variables to be edited,
- *HEADER* : Title of the restitution which we want to obtain.

In the case of the example above,

- *ZONES* would equal: 6 1 2 3 5 6
- *HEADER* would be the array:

REG No.	NAME	AGE	SEX	M.S.	REG No.

THE CLIENT

To provide comercial follow up and invoicing, a company has to be able to organise, for each of its clients:

- a client code, of four figures maximum
- a delivery address, of 4 lines of 30 characters,
- an invoicing address, of the same dimensions.

This information presents certain features:

- In most cases, the invoicing address is identical to the delivery address.
- Some clients have formed buying groups. This means that one of them centralizes all the invoices concerning a number of members. Each of these members has therefore a different delivery address, but shows the address of their buying group as invoicing address.

DATA STRUCTURE

We could store two addresses for each client. The result of this would be a slight waste of space, since both these addresses would be identical most of the time. Moreover, when a buying group changed address, all its members' invoicing addresses would have to be changed.
We have chosen another approach which you should use.

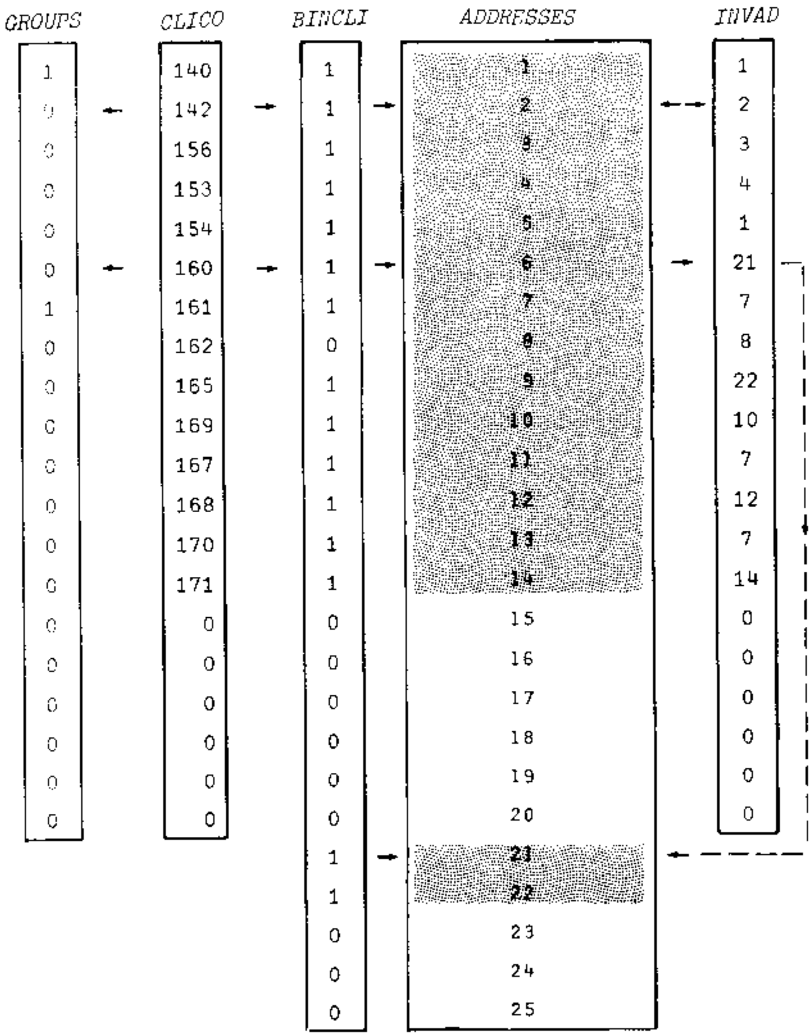
This data structure is presented not because it is the best, but because it leads you to discover methods of organising data which are particularly useful in APL.

Moreover, we have attributed great importance to computing time savings, and to reliability of the system in case of an incident.

The diagram on the following page shows the organisation adopted.

- 1 - All the data is over-dimensioned, so as to contain present clients, and to allow recording of further clients in the near future.
For the reasons for this choice, which appear paradoxical in APL, consult the answers. You will discover that this technique offers excellent reliability and it is much more economical than would appear.

D A T A S T R U C T U R E



You will notice that client 162 has left; there is a zero in *BINCLI*. The parts on a grey background represent addresses which have been entered. The numbers printed in *ADDRESSES* serve only to indicate the lines, so as to facilitate reading of the diagram.

- 2 - A vector, *BINCLI*, will indicate the places occupied by 1's, and the places still unoccupied or left vacant by the departure of a client by 0's.
The new clients will be inserted in the first place found vacant. As a result they will be arranged in any order.

- 3 - The vector *CLICO* contains the clients' codes, not arranged in order, and zeros in the part which is still unoccupied.

- 4 - A single matrix, *ADDRESSES*, will contain all the addresses, at the rate of one address per line. The 4 lines of 30 characters of each address will thus be placed end to end, so as to be stored in a single line of 120 characters.

  The entire upper part of the matrix will be reserved for delivery addresses, each delivery address being positioned opposite the client to which it refers.
  The lower part of the matrix contains the invoicing addresses which differ from the delivery address. Hence there is no longer any correspondence between the client's index and the index of his invoicing address.

- 5 - The vector *INVAD*, of the same length as *CLICO*, contains for each client the index of the place where his invoicing address is located.

  For example:

  Clients 140, 142, 156 and 153 are situated in positions 1, 2, 3 and 4 respectively. Their delivery addresses are thus situated in lines 1 2 3 4 of *ADDRESSES*. In *INVAD* we can read that their invoicing addresses are also in 1 2 3 4.
  For these clients, their two addresses are identical.

  Client 106 is in the 6th position and so is his delivery address. On the other hand, it can be seen in *INVAD* that his invoicing address is the 21st. Hence it is different.

  Finally we can see that clients 167 and 170 have the seventh invoicing address: they are two members of buying group 161.

- 6 - The binary vector *GROUPS* indicates by 1's the clients which are head offices of buying groups. This is so for clients 140 and 161.
The members of these goups will be referenced by another means.

It should be noted that *BINCLI* has as many elements as *ADDRESSES* has lines.

EXAMPLE OF NEW CLIENTS INPUT

```
                    INTROCLI
----> ENTRY FOR CLIENT 349                                    (1)
DELIVERY ADDRESS
     THE OLD OAKS
     29, PRIORY STREET                                        (2)
     LAKESIDE INDUSTRIAL ESTATE
     SHEFFIELD
BUYING GROUP ... 348                                          (3)
----> ENTRY FOR CLIENT 350
DELIVERY ADDRESS
     CAMDEN TOWERS
     17, WINSTON CHURCHILL ROAD

     SOUTHAMPTON
BUYING GROUP ...                                              (4)
INVOICING ADDRESS
     CAMDEN TOWERS
     ADMINISTRATIVE HEADQUATERS                               (5)
     207, SMITH STREET
     SOUTHAMPTON
----> ENTRY FOR CLIENT 351
DELIVERY ADDRESS
     DORIAN FURNISHINGS
     110, St. JAMES ROAD
     PICCADILLY
     LONDON W.1.
BUYING GROUP ...
INVOICING ADDRESS                                             (6)

----> ENTRY FOR CLIENT 352
DELIVERY ADDRESS
     CLEO SHOPS
     88, QUEENS AVENUE
     RUSSEL SQUARE
     LONDON W.C.1.
BUYING GROUP ... 352                                          (7)
----> GROUP RECORDED
----> ENTRY FOR CLIENT 353
DELIVERY ADDRESS

----> END                                                     (8)
```

FIRST STEP

Write the functions necessary for entering new clients. These functions behave as demonstrated by the example set out on the previous page.

1 - The computer attributed the client's codes, in increasing order, 1 by 1.

2 - We indicate the client's delivery address. A blank address (carriage-return) will indicate the end of the task (8).

3 - If the client belongs to a buying group, we enter the code of the latter. The computer automatically deduces its invoicing address and therefore does not ask for it.

4 - On the other hand, a blank reply to the question "*BUYING GROUP*" indicates that the client does not belong to a buying group. Hence the computer asks for his invoicing address. Two alternatives are thus presented

   5 - if we type an address, it will be taken as the invoicing address.

   6 - On the other hand, a blank reply (carriage-return) signifies that the invoicing address is identical to that of delivery.

It is seen that this procedure makes it possible for only one question to be answered for the majority of clients, in order to enter the delivery address.

To create a buying group it has been customary to answer of the question "*BUYING GROUP*" by its own client code (see 7). The computer confirms creation of the group.

SECOND STEP

A function should print-out the list of clients, according to the presentation on page 68, in increasing order of clients code.
We will see:

- the client's code,
- the delivery address,
- the invoicing address, or the statement "*SAME ADDRESS*" if this is the case.

The symbol "*G*" at the begining of a line serves as reference to the buying groups.

For the members of a group, we print, in place of the invoicing address, the statement "--*GROUP*--", followed by the name of the buying group.

The function *PRICLI* performs this task:

PRICLI

G	CODE	DELIVERY ADDRESS	INVOICING ADDRESS
G	345	*SUN KING* 350, *SMITHFIELD ROAD,* *HOLLINGTON,* *LEEDS.*	*SAME ADDRESS*
	349	*THE OLD OAKS* 29, *PRIORY STREET* *LAKESIDE INDUSTRIAL ESTATE* *SHEFFIELD*	-- *GROUP* -- *SUN KING*
	350	*CAMDEN TOWERS* 17, *WINSTON CHURCHILL ROAD* *SOUTHAMPTON*	*CAMDEN TOWERS* *ADMINISTRATIVE HEADQUARTERS* 207, *SMITH STREET* *SOUTHAMPTON*
	351	*DORIAN FURNISHINGS* 110, *St. JAMES ROAD* *PICCADILLY* *LONDON W.*1.	*SAME ADDRESS*
G	352	*CLEO SHOPS* 88, *QUEENS AVENUE* *RUSSEL SQUARE* *LONDON W.C.*1	*SAME ADDRESS*

THIRD STEP

There must be a function which enables a client to be erased, but only after having checked that we have not committed a code error. For this, the function must display the clients name and ask for validation (see example on following page).

It must be impossible for the cancelled clients code to be used by a new client.

When the cancelled client is the head office of a buying group, a message must clearly indicate the other clients who were members of this group. Moreover, the function must correct the invoicing address of these clients in such a way that they are henceforth invoiced at their delivery address.

The funciton *ERASECLI* will perform this task:

```
            ERASECLI
      CLIENT CODE
      ⎕:
            389
(1)   WOOD SORREL                        / VALIDATE....... NO
      CLIENT CODE
      ⎕:
            398
(2)   VULCAN'S FORGE                     / VALIDATE.......
      ----> CLIENT ERASED
      CLIENT CODE
      ⎕:
            345
      SUN KING                           / VALIDATE.......
(3)   THIS BUYING GROUP INVOLVED CLIENTS: 349 356
      ----> CLIENT ERASED
      CLIENT CODE
      ⎕:
            END
```

NOTES

1 - After entry of the client's code, the computer displays its name, and requests validation. Here there has been a code error (389 instead of 398), and the user replies NO. Hence this client will not be erased.

2 - This time the code was correct, so the user replies YES, or even nothing at all as here.

3 - When a buying group disappears, this message notifies the user, while displaying the clients who were members of it.

A final step would consist of writing functions for correcting addresses. The data structure used here would complicate this task to a rather significant degree. This is its main shortcoming.

BLOCK-HEADS

As in the topic *"Block and Tackle"*, we have an array of *L* lines and *C* columns, representing *C* successive values of *L* bits of information.
For example, *RESULTS* represents the results of the three sales outlets of a wheels manufacturer over the last 6 months:

RESULTS

```
921  668  625  346  800  643
768  778  440  942  907  924
365  398  616  711  780  830
```

This time, we are concerned not only with the results of each sales outlet, but also with the monthly totals. The presentation of the diagram will thus be slightly different.

The vertical scale and the width of the blanks will be controlled by the left-hand argument of the plotting function.
For example, with the scale 5 in one thousand in height, and with blocks of 8 characters width, the following is obtained:

0.005 8 *BLODIAG RESULTS*

```
                                |-------|
                                |       |-------|
|-------|               |-------|       |       |
|       |               |       |       |       |
|       |-------|       |-------|-------|-------|
|       |       |-------|       |       |       |
|       |       |       |       |       |       |
|-------|-------|       |       |       |       |
|       |       |-------|       |       |       |
|       |       |       |-------|-------|-------|
|       |       |-------|       |       |       |
|-------|-------|       |       |       |       |
|       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|
```

Furthermore, we wish to be able to omit the width of the blocks; for example:

0.01 *BLODIAG RESULTS*

In default the funciton must plot blocks 5 characters wide.

OIL ON TROUBLED WATERS

We want to follow the cost prices of a range of lubricants. Three variables contain the following pieces of information:

- the vector *CODE* contains the codes of each of these products,
- the matrix *PRODUCT* contains descriptions of them, of 20 characters.
- Finally the vector *PRICE* contains the cost prices.

Certain products are basic products, whose cost price is known. Others are mixtures. Their price is therefore calculated from the proportion of the qualitites of products used in their composition.

Hence part of the vector *PRICE* is entered manually, whereas the rest is calculated automatically.

In the following examples, we have used a mini catalogue of imagined products, which adequately covers all the possible alternatives.

You are asked to write a set of functions enabling the functions shown hereafter to be created:

- 1 - PRINT-OUT OF THE COMPLETE CATALOGUE

PRINPROD

CODE	*PRODUCT*	*PRICE*
207	*CASTOR OIL*	60
208	*ELBOW GREASE*	106
209	*SARDINE OIL*	20
210	*DO. 20 W 40*	68
215	*FRYING OIL*	33
318	*EMULSIFIER E*	300
320	*SO. 20 W 40*	62
332	*PEANUT OIL*	150
333	*VINEGAR*	52
335	*VINAIGRETTE*	148

This catalogue is printed in increasing code order with prices shown.

- 2 - ENTRY OF NEW PRODUCTS

We will in fact introduce only their description; the computer will assign the codes, in increasing order, starting with the biggest. The prices and compositions are introduced at a later stage.

Below, we have entered two new products. The code on the left is printed by the computer and the description is supplied by the user. The program ends when the user does not supply a description.

INTROPOD

CODE	*DESCRIPTION*	
336	*COD-LIVER OIL*	The code assigned is
337	*SPECIAL COMPETITION*	actually equal to 335 + 1.
338		

We could check the result by printing a new catalogue. We would find that the prices of these products are temporarily zero.

- 3 - INTRODUCING OR UPDATING COMPOSITIONS

This task is performed in two stages:

a) We enter the code of the product concerned and the computer displays its description as a means of checking. It is the computer's job to confirm that there is no error before proceeding to the 2nd stage.

b) We enter the composition in the form of a series of pairs of numbers. For example, in the following processing we typed:

5 210 30 336 45 320 20 335

This signifies that the product concerned (337) is composed of:

5% of product 210
30% of product 336
45% of product 320
20% of product 335

Of course, the computer checks that we have introduced pairs of values and that the codes introduced actually exist.
On the other hand, it does not check if the total makes 100%.

Example:

```
            UPDCOMP
       PRODUCT CODE [END]
       ⎕:
            337                                        1
       SPECIAL COMPETITION ; VALIDATE...               2
       COMPOSITION [END]                               3
       ⎕:
            5 210 30 336 45  320  20  335              4
       PRODUCT CODE [END]                              5
       ⎕:
            END
```

In (1), we enter the product code. The computer displays its name (2), and requests confirmation. The user can, as here, answer nothing, and the processing continues, or answer *NO* if he thinks he has committed a product code error. In such cases the computer would put the question again.
In (4) we enter the product composition, as shown on the preceding page. We then pass on to the next product. We type *END* in order to stop

We will assume that a product NEVER comprises more than 10 components.

When this updating is carried out, the constituted products prices are re-calculated, in accordance with the new compositions.

- 4 - PRINTING THE COMPOSITION OF DERIVATIVES

We will adopt the form below, where each product name is followed by its composition. For example, product 320 is composed of 85% of product 209, and 15% of product 318.

```
      PRINCOMP
CODE    PRODUCT                 COMPOSITION

210  DO. 20 W  40          83 P 207 + 17 P 208
320  SO. 20 W  40          85 P 209 + 15 P 318
335  VINAIGRETTE            3 P 318 + 90 P 332 +  7 P
337  SPECIAL COMPETITION    5 P 210 + 30 P 336 + 45 P + 20 P 335
```

- 5 - UPDATING THE LIST OF PRICES

The computer asks which prices are to be modified. If we reply *ALL*, the computer seeks all the basic products and asks their price. If, as below, we give a restricted list of products, the computer checks that it concerns basic products, then asks their price, one after the other.

For simplification, reference can be made to the current price of a product. Hence the expression *SAME*+15 signifies that the price of product 318 is increased by £15.

```
        UPDPRICE
WHICH PRODUCTS [ALL, END]
⎕:
        209 320 332 336
----> INCORRECT: 320                  This is actually a compound.
WHICH PRODUCTS [ALL, END]
⎕:
        209 318 332 336
SARDINE OIL                           The computer displays the 4
⎕:                                    products, and asks their price.
        19
EMULSIFIER E 410
⎕:
        SAME+15                       Here reference is made to the
PEANUT OIL                            present price.
⎕:
        156
CODLIVER OIL
⎕:
        41
```

All the prices of derivatives are recalculated of course in the same way. We can see this by printing the new catalogue of products *PRINPROD* (see next page).

It is seen in this print-out that the prices of products 209 318 332 and 336 have been modified, and that the prices of products 320 and 337 have been calculated on these bases.

CODE	*PRODUCT*	*PRICE*
207	*CASTOR OIL*	60
208	*ELBOW GREASE*	106
209	*SARDINE OIL*	19
210	*DO.* 20 *W* 40	68
215	*FRYING OIL*	33
318	*EMULSIFIER E* 410	315
320	*SO.* 20 *W* 50	64
332	*PEANUT OIL*	156
333	*VINEGAR*	52
335	*VINAIGRETTE*	154
336	*CODLIVER OIL*	41
337	*SPECIAL COMPETITION*	76

- 6 - ERASING A PRODUCT

Before erasing, the computer displays the product name and requests confirmation, as with step 3.
If the product is used in the composition of another product, the computer must refuse to erase it.

```
        ERASEPROD
PRODUCT CODE [END]
□:
        318
EMULSIFIER E 410      ;VALIDATE...
THIS PRODUCT IS A COMPONENT OF: 320 335
----> IT MAY NOT BE ERASED
PRODUCT CODE [END]
□:
        215
FRYING OIL            ;VALIDATE...      ←
PRODUCT CODE [END]
□:
        END
----> TERMINATE
```

This product does not occur in the composition of other products; it will be erased.

- 7 - CHANGE OF STATE

It could happen that a derivative, manufactured by the company, is replaced by an identical product bought ready made from a sub-contractor. Hence it passes from the "derivative" state to the "basic product" state. A function must allow this passage, and enquire the new cost price of the product. Here again, we will be able to make reference to the current price.

```
            ERASECOMP
PRODUCT CODE [END]
⎕:
            210
DO. 20 W 40             ;VALIDATE...
NEW PRICE [SAME]
⎕:
            SAME-18
PRODUCT CODE [END]                       The computer links on to another
⎕:                                       product.
            END
----> TERMINATE
```

Again, all the prices will have to be recalculated.

Here is the state of the data after all these processings:

207	*CASTOR OIL*	60
208	*ELBOW GREASE*	106
209	*SARDINE OIL*	19
210	*DO. 20 W 40*	50
318	*EMULSIFIER E* 410	315
320	*SO. 20 W 40*	64
332	*PEANUT OIL*	156
333	*VINEGAR*	52
335	*VINAIGRETTE*	154
336	*CODLIVER OIL*	41
337	*SPECIAL COMPETITION*	75

It is your task to write all these functions, and to define an adapted data structure.
Do not try to over-optimise your answer. For example, after each price or composition modification do not hesitate to recalculate ALL the derived prices if this can simplify your function. It is understood that in an actual application bearing on several hundred products one would only recalculate the prices affected by the modifications carried out.

THREE PUZZLES

We will finish with three puzzles the sole purpose of which is to entertain us. To increase the difficulty, they will have to be solved WITHOUT LOOPS.

THEME 1

We wish to transform a vector of characters by inserting a number of blanks into it so that the resulting vector has a prescribed length. This is what we call "justifying" a text. Imagine that *RECALL* is the following vector:

```
RECALL ← 'CREATING LOOPS IS TOTALLY PROHIBITED'
```

This is a vector of 36 characters. It can be increased to 40 characters by:

```
      40 JUSTIF RECALL
CREATING  LOOPS  IS  TOTALLY  PROHIBITED
```

It can be increased to 45 characters by:

```
      45 JUSTIF RECALL
CREATING    LOOPS   IS   TOTALLY   PROHIBITED
```

The blanks inserted by supernumbering are divided as uniformly as possible at places where the initial text already had blanks.

Such a function enables texts to be presented in a particularly aesthetic way. We could have used it to improve the result of the function *PRINTEXT* in the topic "*Now throw away my book*"

THEME 2

Starting with any vector, we wish to constitute a second from it by repeating each of its elements. The number of repetitions is given by the left-hand argument of the function; the vector to be transformed is given as right-hand argument:

```
      3 2 5 1 6 REPRO 2 8 6 5 4
2 2 2 8 8 6 6 6 6 6 5 4 4 4 4 4 4
```

In accordance with the left-hand argument, we indeed obtain:

3 times the value 2,

2 times the value 8,

5 times the value 6, etc...

Preferably, the function will have to accept vectors of characters as right-hand argument:

```
      1 2 2 REPRO 'ROT'
ROOT
```

THEME 3

We know the differences which separate various points in a town. These points are numbered (here from 1 to 6), and we have assigned the distances which separate them in the matrix *DIST*:

	1	2	3	4	5	6
1	0	6	7	4	6	7
2	6	0	2	5	10	4
3	8	2	0	5	9	3
4	3	11	5	0	6	4
5	6	10	9	5	0	8
6	7	4	4	4	8	0

Hence, the distance from point 1 to point 3 is equal to 7, whereas the distance from point 3 to point 1 is equal to 8.

We wish to know the length of any journey (closed or not) which is given by the list of points encountered. A function must undertake this calculation:

```
      DIST BUS 2 5 1 3 6
26
```

This signifies that the journey 2 → 5 → 1 → 3 → 6 is of length 26.

We can generalise the problem to accept a matrix of journeys comprising the least stages by immobile stages:

```
2 5 1 3 6 6 6
4 1 6 5 2 3 4
1 3 2 4 5 6 6
```

TO THE READER

This work is certain to contain errors, gaps and imperfections. The author would welcome the benefit of your criticisms, suggestions or answers, if they are clearer. They will serve to improve later editions and to create an exchange of views, which can only be advantageous to the development of APL.

Do not hesitate to send your detailed comments to the publisher, in the following form, so that we know who you are and can answer you if need be.

Thanking you in advance.

Bernard LEGRAND

WORK "*APL: Management Problems with Answers and box of tools*"

COMMENTS for the author, sent by:

Mr.

Company:

Address:

Telephone:

Envelopes adressed to: John Wiley & Sons Limited,
Baffins Lane,
Chichester,
Sussex.
PO19 1UD.

A N S W E R S

References between square brackets [8] refer to instructions of the functions described in the text.

References to the course refer to the work "Learning and Applying APL", by the same author.

PRELIMINARY WORK

FIRST THEME

A simple method consists of generating a series of whole numbers, the beginning of which is then dropped:

```
        ∇ R←BEGIN TO END
[1]     R←(BEGIN-1)↓⍳END
        ∇
```

This method works of course, but if we want to calculate the whole numbers from 5629 to 5644 it is unnecessary to create a series of more than 5000 values in order then to retain only a few of them!

Hence the following method is preferable:

```
        ∇ R←BEGIN TO END
[1]     R←(BEGIN-1)+⍳END-BEGIN-1
        ∇
```

This time only the necessary whole numbers are generated.

GENERALISING

If we generalise the function, we see that it receives operands which this time are vectors.
For example:

```
41  52  56  to  65  77  83  98 ....etc....
```

In such an expression, the series of whole numbers to be created is determined uniquely by the last term of the left-hand operand, and the first of the right-hand operand. The other elements are conserved without change.
Whence the solution:

```
        ∇ R← BEGIN TO END;D;SERIES
[1]     SERIES←D+⍳(1↑END)-D←¯1↑BEGIN
[2]     R←BEGIN,SERIES,1↓END
        ∇
```

Other very colse alternatives are possible.

SECOND THEME

This function presents no difficulty.

```
        ∇ R←ADDRESS AFTER MESSAGE
[1]       ''
[2]       '---->',MESSAGE
[3]       R←ADDRESS
        ∇
```

THIRD THEME

We cannot place a vector vertically by transposition, which has no effect on vectors. Hence it must be transformed into a matrix.

If the vector has n elements, a matrix of n lines and a single column must be made from it, i.e:

```
((ρVEC),1) ρ VEC
```

Hence we can apply the FORMAT function, which gives the following function:

```
        ∇ R←FOR VERT VEC
[1]       R←FOR ⍕ ((ρVEC), 1) ρ VEC
        ∇
```

We can improve this first draft by establishing that the second element of the FORMAT is often zero (print-out of whole numbers). Hence the user can be spared the trouble of explicitly indicating it by means of the TAKE function.

In effect : 2 ↑ *FOR* will give 6 2 if the user has typed 6 2

but : 2 ↑ *FOR* will give 6 0 if the user has typed 6

Hence the following function is obtained:

```
        ∇ R←FOR VERT VEC
[1]       R←(2↑FOR) ⍕ ((ρVEC),1) ρ VEC
        ∇
```

FOURTH THEME

The relative spacing between two data *A* and *B* is obtained by one of the two following formulae:

```
(A-B)÷B    or    (A÷B)-1
```

The rsult obtained is decimal. Its percentage value can be obtained by multiplying by 100, then rounding off the result.

```
        ∇ R← A PC B
[1]       R←(A÷B)-1
[2]       R←⌊0.5+100×R
        ∇
```

This function accepts operands of all dimensions.

FIFTH THEME

There is an apparent answer: laminating. This is the short, elegant answer in appearance.

```
        ∇ R← A WITH B
[1]       R←A,[1.5]B
        ∇
```

Certain functions however, will probably require that we prepare a matrix for them constituted by linking several vectors as right-hand operand. For example:

```
FUNC V1 WITH V2 WITH V3 WITH V4
```

Hence, while the first function *WITH* indeed receives two vectors as operands, the second receives a matrix. The preceding answer falls by the wayside. Hence we will retain the following solution for preference, which has the elegance of solutions which work well:

```
        ∇ R←  A WITH B
[1]       R← A, (2↑(ρB),1)ρB
        ∇
```

SETTLING OF ACCOUNTS

FIRST STEP

The easiest method consists of calculating all the values which must feature in the chosen column of *BOOK*, and catenating them into a vector of 14 values.

We thus calculate:

- the costs, which are the first eight values introduced, and their total (instruction [1]),
- the returns, which are the last three values of the right-hand argument, and their total (instruction [2]),
- the month's result, which is the total returns minus the total costs.

All these values are then catenated into a vector, which therefore indeed possesses 14 elements [4].

It is sufficient to place these values in the column of *BOOK* which is designated by the left-hand argument; instruction [5].

```
     ∇ MONTH RECEIVES VALUES ;TOTCOS;COS;TOTRET;RET;REC;COL
[1]    TOTCOS ← +/ COS←8↑VALUES
[2]    TOTRET ← +/ RET←¯3↑VALUES
[3]    REC ← TOTRET-TOTCOS
[4]    COL ← COS, TOTCOS, RET, TOTRET, REC
[5]    BOOK[;MONTH] ← COL
     ∇
```

Here we have used the function TAKE; we could have equally well extracted the values by indexing:

COS ← *VALUES*[⍳8] and *RET* *VALUES*[9 10 11]

It must not be forgotten to place, in the header, all the variables which are only working variables internal to the function. On the other hand, the function acts directly on *BOOK*, which is a GLOBAL variable for it.

The function does not give an explicit result, its only aim being to update an array of values, and not to restore a result destined for other calculations.

IMPROVEMENTS AND DETAILS

Execution of the function produces no apparent effect. It is thus advisable, in such cases, to print-out a message indicating that the function has suitably run its course.
For example:

IT'S DONE

Furthermore, it would be sensible to check that the data entered satisfies certain essential conditions:

- only one month must be entered,
- this must be a value between 1 and 12,
- it is essential that 11 values are entered.

All these checks must be made before any other work, which would lead to writing the function thus:

```
        ∇ MONTH RECEIVES VALUES ;TOTCOS;COS;TOTRET;RET;REC;COL
[1]       MONTH←1↑MONTH
[2]       VALUES←,VALUES
[3]       →((MONTH∊⍳12)∧(11=⍴VALUES))/OK
[4]       'MONTH OR NUMBER OF VALUES INCORRECT'
[5]       →0
[6]       OK: TOTCOS←+/COS←8↑VALUES
[7]       TOTRET←+/RET←8↓VALUES
[8]       REC←TOTRET-TOTCOS
[9]       COL←COS,TOTCOS,RET,TOTRET,REC
[10]      BOOK[; MONTH]←COL
[11]      'IT'S DONE'
        ∇
```

We can see that all these details have finished by doubling the size of the function.

SECOND STEP

Print-out is achieved very simply by linking the array of characters *NECTAR* and an extract from the numeric array *BOOK*.
Of course, we can attain this only by transforming this extract into an array of characters also, by means of the FORMAT function (⍕).

The columns to be extracted are given by the argument of the function.

We have chosen to use the dyadic FORMAT here, so as to check the position of the columns of numbers very precisely.

```
        ∇ DISPLAY LIST
[1]     (20⍴' ') , 8 0 ⍕ LIST
[2]     ''
[3]     NECTAR , 8 0 ⍕ BOOK[; LIST]
        ∇
```

The header is obtained by printing 20 blank spaces (width of the array *NECTAR*), followed by the list of months, in the same format as the extract of the array *BOOK*.

THIRD STEP

Having extracted the columns of *BOOK* corresponding to the months desired, we can seek the non-zero values. A reduction by OR will give a binary vector, the columns having at least one non-zero value. This is the meaning of the instruction [1], which could equally be written:
0 ∨.≠ *BOOK*[;*LIST*]

By keeping only the useful months from the list of months, we can keep the remainder of the program unchanged.

```
        ∇ DISPLAY LIST ;CACHE
[1]     CACHE ← ∨/[1] BOOK[;LIST]≠0
[2]     LIST ← CACHE/LIST
remainder unchanged:
[3]     (20⍴' '),8 0⍕LIST
[4]     ''
[5]     NECTAR, 8 0⍕ BOOK[; LIST]
        ∇
```

Printing the months clearly is a very common problem. First of all a variable containing the names of the months, complete or abbreviated as here, must be constituted. As this variable can be used with many other functions it will remain GLOBAL in the workspace.

Here it is called *JANDEC*; it is a matrix of 12 lines and 4 columns.

```
JANU
FEBR
MARC
APRL
... etc ...
```

We will assume that the variable *LIST* takes the value: 7 8 9.

The expression *JANDEC*[*LIST*;] will give a matrix of 3 lines and 4 columns, constituted as follows:

```
JULY
AUGT
SEPT
```

We now wish to obtain: *JULY*   *AUGT*   *SEPT*

For this we precede the matrix by as many blank columns as are necessary so that each month's name has the required width. We achieve this either by catenation, or more simply by the TAKE function:

```
((ρLIST), ¯8) ↑ JANDEC[LIST;]
```

By ravelling the result (,) we obtain the vector of the required header.

In the function thus modified, we preferred to generalise the width of the numbers' columns, calling it *FOR*. This format can be either a GLOBAL variable, or a LOCAL variable, if we wish that only somebody familiar with APL is authorised to modify it.

```
        ∇ DISPLAY LIST ;CACHE;DIM
[1]     CACHE ← /[1] BOOK[;LIST]≠0
[2]     LIST ← CACHE/LIST
[3]     DIM ← (ρLIST),-FOR[1]
[4]     (20ρ' '), , DIM↑JANDEC[LIST;]
[5]     ''
[6]     NECTAR, FOR ⍕ BOOK[;LIST]
        ∇
```

Here, *FOR* would have the value 8 0. It is a global variable.

Note, the two commas in line [4]. One is the ravel which enables the matrix of the months' names to be placed in a vector; the other is the catenation of this header into 20 blanks, as previously.

FIVE COLUMNS INTO ONE

The traditional solution to this problem consists of printing the lines one by one, each time comparing the family code to that of the preceding line. With each change of code, we print a header.
In APL, it is possible to know the dimensions of the different "paragraphs" which must be printed immediately.
Each paragraph will be printed as a whole and we will take account of the lines remaining available at the bottom of the page, so as to know if there is enough space for the next paragraph.

Such is the working layout of the solution presented hereafter.

We have retained the following notations:

- *CUTS* — is a vector indicating the numbers of lines of *LISBASKET* after which we change the family code. In other words, this vector indicates the ends of paragrpahs.
- *SIZE* — is the number of lines of *LISBASKET* containing a given paragraph.
- *OCCUPIED* — is the total number of lines occupied by a paragraph. *OCCUPY* of course equals *SIZE*+6.
- *REMAINDER* — is the number of blank lines remaining available on a draft-screen. At the start, *REMAINDER* equals *DRAFT*.
- *ORIGIN* — marks the position of the end of the preceding paragraph. Hence this value serves as origin for calculation of the numbers of lines to be printed.

A preliminary function initialises all these values, and constitutes the variables *HEADER* and *DASHES* which will be used in the presentation.

```
      ∇ PREPARATION ;USELESS
[1]     CUTS ← (LISFACO≠1↓LISFACO,999)/⍳⍴LISFACO
[2]     HEADER ← ' ',,(- +/LISSP=' ')⌽LISSP
[3]     DASHES ← (10×1↑⍴LISSP)⍴'        ---'
[4]     REMAINDER←DRAFT
[5]     ORIGIN←0
[6]     ⍞←'PLACE THE PAPER AT THE TOP OF THE PAGE'
[7]     USELESS←⍞
      ∇
```

Detection of cuts is made by comparison between the vector *LISFACO* and the same vector displaced one position.
If *LISFACO* is the vector:

```
                                              1 1 1 1 2 2 2 2 2 2 3 3 4 4 4 4 5 ......
CUTS will equal ........:                           4           10  12      16    ......
```

In order to improve presentation of the names of the towns, we have sought to frame them on the right, and not on the left. For this we count the number of blanks in each line by +/*LISSP*=' ', and we make them pass to the top by rotation (see a better method on page 183 of the course). By ravelling this matrix, we obtain a vector of the suitable header. Here we have preceded it by two blanks so as to centre it better [2]. This explains the double comma: one to ravel the matrix, the other for catenation.

The lines of dashes will be printed by means of a variable calculated once for all [3].

At the start of the work, the number of lines remaining blank is of course equal to a draft [4], provided that the paper is suitably positioned. Hence a message asks the user to position his paper. But the computer must wait for the end of this manoeuvre. This is why we hand over to the user by means of a quote-quad [7]. No use will be made of what the user answers, whence this variable *USELESS*.

PRINCIPAL FUNCTION

After preparation, a loop prints the paragraphs one by one. It is the function *PRESENTATION* which effectively carries out the printing:

```
        ∇ SPLIT;SIZE;HEADER;DASHES;REMAINDER;ORIGIN;CUTS;OCCUPIED
[1]       PREPARATION
[2]       LOOP: OCCUPIED←6+SIZE ← CUTS[1]-ORIGIN
[3]       PRESENTATION
[4]       REMAINDER←REMAINDER-OCCUPIED
[5]       ORIGIN←CUTS[1]
[6]       CUTS←1↓CUTS
[7]       →(0<⍴CUTS)/LOOP
        ∇
```

The size of the first paragraph is given by the first value of *CUTS* reduced by the origin [2], knowing that we will reduce *CUTS* bit by bit with each paragraph-printing [6].

The value of *OCCUPIED* is deducted immediately from this.

We can then print the paragraph [3].

There now remain only *REMAINDER-OCCUPIED* lines available at the bottom of the page [4].

It is fitting next to prepare printing of the following paragraph. If the paragraph which has just been printed ended with the 10th line of *LISBASKET*, this value will serve as origin for calculating the indices of the next paragraph. This is the meaning of instruction [5].

We can amputate *CUTS* of its first value and pass to the following paragraph, if there is one left, of course [7].

PRESENTATION AND PRINTING

To print a paragraph is fairly simple if one respects the instructions of the statement.
The lines to be printed are given by *ORIGIN*+ ⍳*SIZE*, and the family number by *LISFACO*[*ORIGIN*+1].

```
      ∇ PRINT-OUT ;EXTRACT
[1]     ''
[2]     HEADER,'  ',LISFA[LISFACO[ORIGIN+1] ;]
[3]     ''
[4]     10 0 ⍕ EXTRACT←LISBASKET[ORIGIN+⍳SIZE] ;]
[5]     DASHES
[6]     10 0 ⍕ +/[1] EXTRACT
[7]     ''
      ∇
```

We have named the portion of the matrix printed in [4] *EXTRACT*. After printing a line of dashes [5], it suffices to print the whole of *EXTRACT*, in the same format [6].

For the header the variable *HEADER* contains the names of the selling points. It suffices to follow it by the family name. We obtain this by extracting the line corresponding to the family printed [2] from the matrix *LISFA*.

Now all that remains is to verify the page setting.

An initial test serves to determine if the remaining space is sufficient for printing the subsequent paragraph [1]. If *OCCUPIED* is less or equal to *REMAINDER* on the page, we can go to instruction [5] and proceed with printing.

```
      ∇ PRESENTATION
[1]   →(OCCUPIED≤REMAINDER)/VABENE
[2]   →(REMAINDER=0)/TOPOFPAGE
[3]   (REMAINDER-1)ρ⎕TC [3]
[4]   TOPOFPAGE: REMAINDER←DRAFT
[5]   VABENE: PRINT-OUT
      ∇
```

If there is not enough space left, we must pass as many blank lines as remains at the bottom of the page, in order to start the next paragraph at the top of a page.
We achieve this by printing (*REMAINDER*,1)ρ' ', i.e. a blank matrix of *REMAINDER* lines.
More often, we use the character "line feed" which we obtain by indexing the atomic vector ⎕ or the terminal control characters vector ⎕*TC* (see course page 277). It is this solution which has been retained in instruction [3] .

But take care, printing of *n* line feeds actually causes *n* +1 blank lines on the paper, since even printing of an empty vector causes a blank line to be printed. This is why we have not printed *REMAINDER* line feeds but *REMAINDER*-1.

Another essential precaution: having jumped the appropriate number of lines, we are positioned at the top of a page, and it is advisable to update *REMAINDER* [4].

Finally, if by chance the preceding paragraph printed ends exactly at the bottom of its page, without leaving any space, it is pointless to feed blank lines. This is the meaning of instruciton [2], which in such cases returns directly to *TOPOFPAGE*.

The solution presented in the preceding pages admits numerous alternatives of detail. It is important to arrive at a good result. However, you will notice that a wise choice of variable names enlightens the reader, and greatly facilitates reading and understanding of the programs.

SELECTED INVOICES

FIRST STEP

Having introduced the list of required months by means of a QUAD, it suffices to seek the months of issue belonging to this list. We obtain, in *SEL*, a binary vector of selection.[1 to 3].

DATES[;2]=⎕ would be equally suitable if we selected only one month, but would result in a length error as soon as one wanted to extract several months simultaneously.

The vector *SEL* would suffice, by a series of logical compressions, to extract the values to be printed, but indexing will appear more suitable for use with the following questions, whence instruction [4]. It is found, moreover, that a succession of compressions is infinitely more costly than a succession of extractions by indexing.

```
      ∇ PRINV ;SEL;REC
[1]     ''
[2]     'MONTHS SELECTED'
[3]     SEL←DATES[;2]∈⎕
[4]     SEL←SEL/⍳⍴SEL
[5]     ''
[6]     'MONTH  DAY  AMOUNT    PAYMENT'
[7]     ''
[8]     REC←DATES[SEL;2 1],AMOUNTS[SEL],DATES[SEL;3 4]
[9]     4 0 5 0 9 0 9 0 5 0 ⍕ REC
      ∇
```

Three instructions [5 to 7] serve to present the header.

The result to be displayed is obtained by catenating the various constituents. Note that *DATES*[*SEL*;3 4] is a matrix; also the vector *AMOUNTS* attaches itself vertically (by default) on its left, without the need to transform it into a matrix. Remember that a vector can be catenated to a matrix, provided that their dimensions are coherent.

Be careful, as well, of the order of extraction from the columns of *DATES*: we want the issuing month first of all, then the day.

In instruction [8], a format enables the columns of numbers to be adjusted under the corresponding headers. We could dispense with this and simply print the numeric matrix *REC*, but centering of the header would have been more delicate.

SECOND STEP

We have manually constituted a small matrix of characters called *MONTHS*, of 12 lines and 4 columns, containing the names of the months. It is a GLOBAL variable which will remain in the workspace, and will certainly serve on other occasions:

```
JANU
FEBR
MARC
APRL
MAY

etc...
```

It suffices to index this array by the list of the issuing months in order to obtain a matrix of characters, here called *MS*. This matrix is catenated to the matrix of characters obtained by putting the columns of numbers into format.

There are few instructions to change:

```
       ∇ PRINV ;SEL;REC;MS
[1]    ''
[2]    'MONTH SELECTED'
[3]    SEL←DATES[;2]∊⎕
[4]    SEL←SEL/⍳⍴SEL
[5]    MS←MONTHS[DATES[SEL;2] ;]
[6]    ''
[7]    'MONTH  DAY  AMOUNT   PAYMENT'
[8]    ''
[9]    REC←DATES[SEL;2] , AMOUNTS[SEL], DATES[SEL;3 4]
[10]   MS, 5 0 9 0 9 0 5 0⍕ REC
       ∇
```

Whereas in the preceding step the FORMAT function was not absolutely essential, here we can no longer catenate *MS*, which is a matrix of characters, to *REC*, which is a numeric matrix. It is essential to convert the latter into a characters array either by a monadic FORMAT, or by a dyadic FORMAT, as above.

THIRD STEP

Since an expression typed by means of a QUAD is evaluated, if we give the value ⍳12 to a variable called *ALL* (instruction inserted in [2.5]), the fact of answering *ALL* to the question asked is equivalent to introducing the value ⍳12 into the QUAD. Hence we select all the months. This new variable must be localised in the header.

```
[2.5]   ALL←⍳12
```

The expression 1 2 3 4 5 6 7 8 *EXCEPT* 3 4 5 signifies that from the list (ι8), we keep only the values which do not belong to a list of rejects (3 4 5).
The function is deducted from this immediately.

```
        ∇ R←LIST EXCEPT REJECTS
[1]     R←(~LIST∊REJECTS)/LIST
        ∇
```

As for the function *TO*, this was written at the beginning of this work; it is useless to apply to it any modification whatsoever.

```
        ∇ R←BEGIN TO END
[1]     R←(¯1↑BEGIN) + ι(1↑END)-1+(¯1↑BEGIN)
[2]     R←BEGIN,R,END
        ∇
```

FOURTH STEP

All expressions of the form *AMOUNTS* ≥ 10000 give a binary vector, which will be received by the QUAD.

Now, we already have a binary vector *SEL*, which serves to select the months. By combining these two vectors by a logical AND, we will select the invoices corresponding to the months and amounts required.

We can proceed in two steps:

```
        'AMOUNTS SELECTED'
        S←⎕
        SEL←(SEL∧S)/ιρSEL
```

This is completely useless, and we can write directly:

```
        SEL←(SEL∧⎕)/ιρSEL
```

Defining *ALL* requires more careful study. Since the function is designed now to accept a binary vector in answer to the QUAD, *ALL* must also be a binary value, which will not modify the initial value of *SEL*.
By giving *ALL* the value 1, the expression *SEL*∧1 will indeed give *SEL*.

```
        ∇ PRINV ;SEL;REC;MS;ALL
[1]     ''
[2]     'MONTH SELECTED'
[3]     ALL←ι12
[4]     SEL←DATES[;2]∊⎕
[5]     ''
[6]     'AMOUNTS SELECTED'
[7]     ALL←1
[8]     SEL←(SEL∧⎕ )/ιρSEL
...     the rest remains unchanged.
```

Finally the function *BETWEEN* must also give a binary result. This requires no explanation.

```
        ∇ R←LIST BETWEEN LIMITS
[1]     R←(LIST ≥ LIMITS[1])∧(LIST < LIMITS[2])
        ∇
```

Such a function can serve on numerous occasions.

FIFTH STEP

Instructions [1 to 8] are unchanged, and we put the list of the months relating to each invoice in *MS*; for example:

Assume that the *MS* is the vector 2 2 2 2 3 4 4 4 5 5

We require to keep only the months which differ from the preceding one. For this, we shift *MS* one notch by means of ¯1↓0,*MS* and we compare with *MS*:

```
                    Here is MS        :   2 2 2 2 3 4 4 4 5 5
                    Here is ¯1↓0,MS   :   0 2 2 2 2 3 4 4 4 5
                                          -------------------
And here is BIN ← MS  ≠ ¯1↓0,MS       :   1 0 0 0 1 1 0 0 1 0
```

This vector serves to eliminate the redundant months by compression *BIN⌿MONTHS*[*MS*;] or *BIN*/[1]*MONTHS*[*MS*;] it serves to insert the adequate number of blank lines by means of expansion. Such is the meaning of instruction [11].

Use of the same binary vector to compress and then expand the same data is a classical APL solution; hold it!

```
        ∇ PRINV ;SEL;REC;MS;ALL;BIN
[1]     ''
[2]     'MONTHS SELECTED'
[3]     ALL←⍳12
[4]     SEL←DATES[;2]∊⎕
[5]     ''
[6]     'AMOUNTS SELECTED'
[7]     ALL←1
[8]     SEL←(SEL∧⎕ )/⍳⍴SEL
[9]     MS←DATES[SEL;2]
[10]    BIN←MS≠¯1↓0,MS
[11]    MS←BIN\[1]BIN/[1]MONTHS[MS;]
[12]    ''
[13]    'MONTH  DAY  AMOUNT    PAYMENT'
[14]    ''
[15]    REC←DATES[SEL;1],AMOUNTS[SEL],DATES[SEL;3 4]
[16]    MS, 5 0 9 0 9 0 5 0 ⍕ REC
        ∇
```

The result is worth while!

NOW, THROW AWAY MY BOOK

ENTERING THE TEXT

After printing a message [1-3], the user is permitted to introduce a line of text by means of a QUOTE-QUAD [5].
If this line is empty (carriage-return), a test [6] returns to the end of the program.

```
      ∇ VEC ← INTROTEXT ;LINE
[1]     ''
[2]     'TYPE YOUR TEXT ; END BY CARRIAGE-RETURN'
[3]     ''
[4]     VEC ← ''
[5]     VAZYTOTO: LINE←,⍞
[6]     →(0=⍴LINE)/OUTPUT
[7]     VEC←VEC,' ',LINE
[8]     →VAZYTOTO
[9]     OUTPUT: VEC← 1↓VEC
      ∇
```

If the line is not empty, we catenate it to the embryo of the already constituted result, separating them by a blank [7]. Then a jump [8] returns to entry of the next line.

In order, however to be able to start the process, an initial value must be given to *VEC*. This is the meaning of instruction [4], where *VEC* receives an empty vector.

Only, when we leave the function, *VEC* starts by a blank which comes from the first execution of line [7]. Hence this blank must be eliminated by instruction [9].

For the meaning of the comma, in line [5], consult the course, page 139.
It is recommended to ravel any value introduced by means of a QUAD or QUOTE-QUAD, if we wish to be able to test its dimensions afterwards.

PRINTING THE TEXT

The aim is to create a matrix which we will constitute little by little, catenating the lines one under the other. As for the vector *VEC* above, an initial value must be given to this matrix *MAT*. This will be a matrix of zero lines, but having the right number of columns, in such a way that the catenations can be made [1].

Next we look for the places where it is possible to cut the text. For this, we examine a section of length *WIDTH*+1. In effect, if by chance it is possible to constitute a line which has exactly *WIDTH* characters, this is explained by the fact that the character situated in position *WIDTH*+1 is a blank.
We have called this part of the text, on which the search [2] is carried out *END*.

```
      ∇ MAT ← WIDTH PRINTEXT VEC ;END;CUTS;POS;LINE
[1]   MAT←(0,WIDTH)ρ''
[2]   SEARCH: END←(WIDTH+1)↑VEC
[3]   CUT←(END=' ')/⍳WIDTH+1
[4]   POS←¯1↑CUTS
[5]   LINE←WIDTH↑(POS-1)↑END
[6]   MAT←MAT,[1]LINE
[7]   VEC←POS↓VEC
[8]   →(0<ρVEC)/SEARCH
      ∇
```

Possible *CUTS* are given by the positions of blanks of the vector *END*. Of course we will take a line which is as long as possible, hence we will keep the last element of *CUTS* as the cutting position.

The blank in position *POS* is useless, since it is at the ènd of a line. Hence, in [5], we keep *POS*-1 characters. However, so that all the lines are of the same length, we adjust them with the *TAKE* function which will add the necessary number of blanks.

The line thus constituted is catenated under the matrix embryo [6].

Thus we can drop the first *POS* characters of the text, including the blank which served in the cutting [7]. If the remaining vector is not empty [8], we continue the process.

IMPROVEMENT

In order to avoid clumsy cuts, it suffices to rectify the vector *CUTS*, without changing the rest of the processing.

In this new perspective, a character enables a cut to be made:

- if it is a blank character,
- and if the following character does not feature among a very precise list of punctuation characters, i.e. ; ? ! :

Two instructions must be changed:

```
[2]  SEARCH: END←(WIDTH+2)↑VEC          to explore one more character.
[3]    CUTS←((¯1↓END)∧~1↓END∊';?!:')/⍳WIDTH+1
```

"HOPE" AND "TRUTH"?

IMBRICATING TWO ARRAYS

A first method consits of reserving a matrix in advance which is the size of the required result, which we fill with anything. There are many ways of achieving this, and here are three of them:

```
R← ((ρA)[1] , 2×(ρA)[2]) ρ 0
R← (1 2 × ρA) ρ 0
R← A , B          answer retained here.
```

Thus we can say that we place the values of the right-hand operand in the equal columns [2], and the values of the left-hand operand in the unequal columns [3]. We obtain:

```
      ∇ R← A IMBRICATES B
[1]   R←A,B
[2]   R[;2×ι(ρA)[2]]←B
[3]   R[;¯1+2×ι(ρA)[2]]←A
      ∇
```

This solution has the disadvantage of accepting only matrices as operands.

A second method consists of "spacing" the columns of the two operands by means of two expansions:

```
      ∇ R ← A IMBRICATES B ;BIN
[1]   BIN ← (2×¯1↑ρA) ρ 1 0
[2]   R←(BIN\A) + (~BIN)\B
      ∇
```

For example, in the case of arrays U and V of the expression, this method constitutes, then adds, the two following sub-arrays:

```
1 0 9 0 5 0 5 0     0 3 0 0 0 3 0 3        1 3 9 0 5 3 5 3
9 0 0 0 0 0 0 0  +  0 6 0 7 0 8 0 8  --->  9 6 0 7 0 8 0 8
1 0 5 0 0 0 5 0     0 8 0 8 0 9 0 6        1 8 5 8 0 9 5 6
```

This procedure is better, because it accepts vectors and arrays of three and more dimensions.

But how can we resist the temptation of a solution using LAMINATION, a little used function and feared by beginners? Of course this solution is completely obscure, but it does not lack a certain charm:

```
        ∇ R← A IMBRICATES B ;DIM
[1]     R←A,[0.5+ρρA] B
[2]     DIM←(¯2↓ρR),×/¯2↑ρR
[3]     R←DIMρR
        ∇
```

In the case of matrices U and V which are of dimensions 3 4, the result of laminating is of dimensions 3 4 2 (see course, pages 137 and 160). Here is its value:

```
1 3
9 0
5 3
5 3

9 6
0 7
0 8
0 8

etc ...
```

Restructuring this array into a matrix of dimensions 3 8 actually gives the result required.

This rather crazy solution also accepts vectors of characters.

COMPARING THE TWO ARRAYS

First of all we must look for columns which contain at least one non-zero value. This can be done by ∨/[1] 0≠*REAL* or also, as in the function considered, by 0∨.≠*REAL*.

Knowing the number of columns to be conserved, we consequently amputate the two arguments, and imbricate them by means of the function written above. This is the role of instructions [1] to [4].

Here we have adapted an asymmetrical format, so that the columns of the result are grouped two by two [5].
Hence we can put the numeric array into format. But the signs columns must be inserted into it afterwards. The last signs column would thus be situated outside the limits of the array. This is the reason why we have catenated a column of blanks on its right [6].

```
      ∇ R ← REAL COMPARE PREV ;DIM;FOR;SPACE;SIGN;COL
[1]     DIM ← +/ 0∨.≠REAL
[2]     PREV ← PREV[;⍳DIM]
[3]     REAL ← REAL[;⍳DIM]
[4]     R ← PREV IMBRICATES REAL
[5]     FOR ← (4×DIM)⍴ 8 0 4 0
[6]     R ← (FOR⍕R), ' '
[7]     SPACE ← 10<|REAL PC PREV
[8]     SIGN ← SPACE× ×REAL-PREV
[9]     SIGN ← '- +'[2+SIGN]
[10]    COL ← 1+12×⍳DIM
[11]    R[;COL] ← SIGN
      ∇
```

The auxiliary function *PC*, written at the beginning of this book, serves to calculate the established spaces. The abnormal spaces are those whose absolute value is greater than 10; instruction [7].

The meaning of the spacing is given by the *SIGN* function (monadic ×). By multiplying these two pieces of information one by the other, we obtain a matrix called *SIGN*, composed:

- of zeros everywhere where the recorded spacings are less than 10%
- of 1 where the results exceed the forecasts by more than 10%
- of ¯1 where they are less than the forecasts by more than 10%.

By adding 2, we obtain, for the data of the expression:

```
2 1 2 2
2 2 3 1
2 3 2 2
```

By indexing a vector of three characters by this matrix [9], we obtain an array of 4 columns containing either blanks, or +'s. or -'s. It suffices to insert these four columns into the appropriate columns of the result [11].

The indices of these columns are calculated in terms of the format adopted in instruction [5]. Since the columns of numbers occupy 8 and 4 positions respectively, the columns of signs will be separated one from another by 12 positions, the first of them being situated at index 13. This is the meaning of instruction [10].

EVERYTHING IS IN THE PRESENTATION

We start by bordering the array on the right and on the left by a blank column in such a way that the horizontal lines are plotted these at the same time as in the rest of the array [1].

POSLI and *POSCO* receive the parameters concerning the lines and the columns to be plotted [2 and 3].

We then attempt to form a binary vector (*BIN*), which will serve in [7] to separate the lines of the given array by an expansion.

- the length of this vector is given by the number of lines of the array, increased by the number of lines to be plotted [4],
- the first horizontal line will be in position 1+*POSLI*[2], but situated AFTER the line of data *POSLI*[2],
- there will be *POSLI*[1]-1 other lines, separated from each other by 1+*POSLI*[3], whence calculation of the positions of lines to be inserted (instruction [5]).
  The new value obtained for *POSLI* serves:
    - to place zeros in *BIN* before expansion [6],
    - to place dashes in *R* after expansion [8].

We can thus frame the array above and below by catenation [9].

```
        ∇ R←LICO PRINT MAT ;POSLI;POSCO;BIN
[1]     R←' ',MAT
[2]     POSLI ← 3↑LICO
[3]     POSCO ← ¯3↑LICO
[4]     BIN ← (POSLI[1]+1↑ρR)ρ1
[5]     POSLI ← +\1+POSLI[2],(POSLI[1]-1)ρPOSLI[3]
[6]     BIN[POSLI] ← 0
[7]     R ← BIN \[1] R
[8]     R[POSLI;] ← '-'
[9]     R ← '¯',[1]R,[1] '_'
[10]    POSCO ← 1+ +\POSCO[2],(POSCO[1]-1)ρPOSCO[3]
[11]    R[;POSCO] ← '|'
[12]    R ← '|',R,'|'
        ∇
```

Calculation of the positions of the columns is very similar, but since it is not necessary to separate the columns of the array, it is pointless to add 1 to the spacing of the columns, such as defined by the argument. On the other hand, the blank column added in line [1] renders it necessary to add 1 to all the positions of the columns [10].
It remains only for the bordering lines to be marked out on the right and on the left [12].

MONTE CARLO METHOD

APERITIF

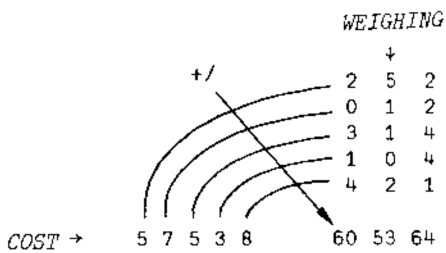
The manufacturing cost of a product is the sum of the products of the prices of the constituent elements by the weighing attributed to them. For the first product, for example, we could write:

```
+/ COST × WEIGHING[;1]
```

But it is more direct to work on all the values by an inner product:

```
COST +.× WEIGHING
```

The figure below reduced to only 3 products and 5 components, clearly demonstrates this calculation:



MAIN COURSE

The number of texts is placed in the left-hand argument. We are going to constitute a matrix of which each line will comprise the 6 production costs calculated for a random sample of prices. This matrix will be called *ASSCOSTS*, and will have *NUMBER* lines, and ¯1↑ρ*WEIGHING* columns [1].

Having made a price supposition for the basic elements [3], we can calculate the 6 production costs by the method shown above. This is the role of instruction [4]; the result being arranged in the 1st line of *ASSCOSTS*.

```
      ∇ R←NUMBER TESTASS ASS ;ASSCOST;SUPPO;I
[1]     ASSCOSTS←(NUMBER, ¯1↑ρWEIGHINGS)ρ0
[2]     I←1
[3]     AGAIN: SUPPO←(ASS[1;]-1)+? ASS[2;]-ASS[1;]-1
[4]     ASSCOSTS[I;]←SUPPO+.×WEIGHINGS
[5]     →(NUMBER≥I←I+1)/AGAIN
[6]     R←(⌊/[1]ASSCOSTS),[0.5](⌈/[1]ASSCOSTS)
      ∇
```

A fine loop compares the index of line *I* with the total number of intended texts [5].
The last instruction seeks the smallest and largest elements in each column of *ASSCOSTS*. We obtain two vectors which we laminate so as to obtain a matrix.

Returning to the random generations, and for this working again on only five basic products, for which the following minimum and maximum assumptions are made:

```
mini →    7   7   8   4   6
maxi →   12  10  20   9  11
```

The range of possible values for each price can be expressed thus:

- for the first : 6 + a value between 1 and 6
- for the second : 6 + a value between 1 and 4
- for the third : 7 + a value between 1 and 13
- etc ....

In other words, the values are obtained by adding, to *ASS*[1;]-1, a random number between 1 and *ASS*[2;]-*ASS*[1;]-1. This is explained by instruction [3].

LET'S FIND A BETTER WAY!

This first solution is quite correct, but was a loop really essential? Could we have found a global approach?
Instead of generating a vector of random prices, we will generate a matrix which has *NUMBER* lines and as many columns as components, for example 10 lines and 40 columns. The product *SUPPO*+.×*WEIGHING* will give a result of 10 lines and 6 columns. We thus obtain directly the 10 generations of the result ... an elegant simplification!

```
      ∇ R←NUMBER TESTASS ASS ;DIM;SUPPO;ASSCOSTS
[1]     DIM←NUMBER,¯1↑⍴ASS
[2]     SUPPO←(DIM ⍴ ASS[1;]-1) + ? DIM ⍴ ASS[2;]-ASS[1;]-1
[3]     ASSCOSTS←SUPPO+.×WEIGHING
[4]     R←(⌊/[1]ASSCOSTS),[0.5](⌈/[1]ASSCOSTS)
      ∇
```

This solution is more concise, as clear as the preceding one and is also executed more quickly, due to the absence of a loop.

THE MOST USEFUL FUNCTION

FIRST DRAFT

```
     ∇ LIST ;FNS;DIM;CR;FUNC;NOS;DASH;ALPHA
[1]   FNS←⎕NL 3
[2]   ALPHA←' ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[3]   FNS←FNS[⍋37⊥ALPHA⍳⍉FNS;]
[4]   DASH←(50⍴'-'),' '
[5]   NEXT: FUNC←FNS[1;]
[6]   DIM←1↑⍴CR←⎕CR FUNC
[7]   TRAIT,FUNC
[8]   ''
[9]   NOS← 2 0 ⍕((DIM-1),1)⍴⍳DIM-1
[10]  NOS←((-DIM),6)↑ '['NOS,']'
[11]  NOS,CR
[12]  ''
[13]  →(0<1↑⍴FNS← 1 0 ↓FNS)/FINISHED
[14]  (30⍴'-'),' PRINT-OUT FINISHED'
     ∇
```

We place the list of names of all the functions in the workspace, given by ⎕*NL* 3 in the matrix *FNS*. Alphabetical classification is carried out by the method shown in statement [3], but an alphabet of 37 letters is needed, followed by decoding in base 37.

The variable *DASH* will serve to separate the functions from each other.

A loop then processes each function individually:

- we take the first function name, and place it in *FUNC*.
- *CR* will contain the canonical representation of this function [6].
- Instructions [7-8] print a line of dashes followed by the function name.
- The lines must then be numbered. They are *DIM*-1 in number since the header is not numbered. In order to place, ⍳*DIM*-1 vertically, it would be possible to use the function *VERT* written at the beginning of this book. For reasons which will be shown later, we have preferred to write the equivalent instruction [9].
- We can frame the numbers with square brackets, and precede them by a blank line and follow them by two blank columns by means of the *TAKE* function [10].
- It remains only to print the numbers list and the canonical representation of the function, side by side [11-12].

We then drop the first name of the list of functions, and proceed to the next if there are still names remaining to be treated [13]. If not we print the end of work message.

IMPROVEMENTS

To eliminate printing of the *LIST* function, it suffices to erase it in the matrix *FNS*. An inner product by ∨.≠ indicates which function names differ by at least one of their letters from *LIST*. In order to accomplish this inner product, the character string "*LIST*" had to be dimensioned to the dimensions of *FNS*. Compression on the lines of *FNS* leaves only the names of the other functions [2].

```
      ∇ LIST ;FNS;DIM;CR;FUNC;NOS;DASH;ALPHA;EMPTY
[1]   FNS←⎕NL 3
[2]   FNS←(FNS∨.≠(¯1↑⍴FNS)↑'LIST')/[1]FNS
[3]   ALPHA←' ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
[4]   FNS←FNS[⍋37⊥ALPHA⍳⍉FNS;]
[5]   DASH←(50⍴'-'),' '
[6]   EMPTY←(0 ,¯1↑⍴FNS)⍴''
[7]   NEXT: FUNC←FNS[1;]
[8]   DIM←1↑⍴CR←⎕CR FUNC
[9]   →(DIM>1)/NORMAL
[10]  EMPTY←EMPTY,[1] FUNC
[11]  →TESTEND
[12]  NORMAL: DASH,FONC
[13]  ''
[14]  NOS← 2 0 ⍕ ((DIM-1),1)⍴⍳DIM-1
[15]  NOS←((-DIM),6)↑ '[',NOS,']'
[16]  NOS,CR
[17]  ''
[18]  TESTEND: →(0<1↑⍴FNS← 1 0 ↓FNS)/NEXT
[19]  →(0∊⍴EMPTY)/NEXT
[20]  (30⍴'-'),' EMPTY OR LOCKED FUNCTIONS'
[21]  ''
[22]  EMPTY
[23]  ''
[24]  OUTPUT: (30⍴'-').' PRINT-OUT FINISHED'
      ∇
```

Most of the function remains unchanged, but at the beginning of the work [6] we constitute an empty array intended to receive the names of the empty or locked functions.

In line [9] a test enables normal processing to be undertaken if the function concerned comprises at least one instruction in addition to its title.

Otherwise we add the funciton's name to the array *EMPTY*, and then jump directly to the end of the work test without printing anything.

At the end of printing, at line [19], if the matrix *EMPTY* is still empty we jump directly to the end of work message [24].

If we find empty or locked functions, a suitable message is printed [20], and we then print the contents of the array *EMPTY* (instructions [21 to 23]).

CRITICISM

The result is a rather long function, which does not summon any sub-function. This is deliberate.
This function is actually intended to be copied from one workspace to another, in order to perform print-outs. It is easier to copy only one function than to copy three or four, disregarding the fact that these functions would then clutter up the workspace.

Among improvements which could still be applied to this function are:

- shifting of labels of a character to the left,
- the control of page setting, in such a way that a function is never printed "straddled" on two consecutive pages. This type of problem is the subject of another topic (see *"Five columns into one"*).
- An alphabetical classification accepting underlined letters, and effective even for very long names.

SEARCHING FOR SKILLED WELDERS

FIRST QUESTION

```
      ∇ WELD1 LIST ;CACHE
[1]   CACHE ← ∨/ WELQUAL∊LIST
[2]   CACHE /[1] WELDERS
      ∇
```

The first instruction enables us, in the matrix *WELQUAL* to examine which qualifications feature in the given list. It is sufficient that a welder possess one OR the other to be displayed. This explains the reduction by OR. The binary cache obtained enables the corresponding names to be selected [2].

SECOND QUESTION

```
      ∇ QUAL WELD2 DATE ;CACHE;MAT
[1]   MAT←WELDATE × WELQUAL=QUAL
[2]   CACHE←∨/MAT>DATE
[3]   (CACHE/[1]WELDERS), 4 VERT CACHE/[1] +/MAT
      ∇
```

WELQUAL=QUAL gives a binary matrix, 1's of which indicate which welders possess the given qualification. By multiplying by the matrix of dates [1], we obtain a matrix *MAT* containing either zeros or the renewal date of the given qualification.

We can thus seek, among these dates, those which are later than the date given as argument [2]. The result is a binary matrix, and the same reduction by OR as before gives the same binary cache.

Extracting the names is similar. Extracting the dates is a little more complex, since they are dispersed in the matrix *MAT*. Here we have added its terms, knowing that there is only one non-zero value per line.
Vertical printing of the dates vector is obtained by the function *VERT* written at the beginning of this work.

THIRD QUESTION

In order to solve this problem, the date indicated by the argument must be compared with the various dates at which the welders must renew their qualifications. These renewal dates can be calculated by adding the validity period to the last renewed date given by *WELDATE*.

Unfortunately, *WELDATE* contains zeros, which will falsify the results. Furthermore, the definitive qualifications are assigned the value zero in the vector *PER*. This must be remedied. For this reason the first two instructions serve to calculate a vector *DURATION*, which is a replica of *PER*, in which the zeros have been replaced by 999.
The instruction [3] gives the renewal dates, adding the validity duration of each qualification to *WELDATE*. We can deal with the zeros of *WELQUAL*, by means of the INDEX-OF function. As there are no zeros in *QUALIF*, the expression *QUALIF*⍳*WELQUAL* gives the value 15 for each zero encountered. On indexing (*DURATION*,999) with 15, we obtain 999.

```
      ∇ WELD3 DATE ;DURATION;NB;NO;SIGNS;URGENT;TOOLATE;REN
[1]     DURATION←PER
[2]     DURATION[(PER=0)/⍳⍴PER]←999
[3]     REN←WELDATE + (DURATION,999)[QUALIF⍳WELQUAL]
[4]     NB←1↑⍴WELDERS
[5]     SIGNS← 2 6⍴'            ---,
[6]     NO←1
[7]     NEXT: URGENT←DATE≥REN[NO;]
[8]     →(~1∈URGENT)/NEXT
[9]     TOOLATE←DATE>REN[NO;]
[10]    WELDERS[NO;], 6 0 ⍕ URGENT/WELQUAL[NO;]
[11]    (10⍴' '), ,SIGNS[1+URGENT/TOOLATE
[12]    OUTPUT: →(NB≥NO←NO+1)/NEXT
      ∇
```

Hence *REN* contains the date at which each qualification of a welder must be renewed. *NB* will contain the number of welders [4]. We then examine each welder in succession, by means of the index *NO*, which is initialised at 1 in [6] and which increases to *NB* in [12].

For each welder, the vector *URGENT* indicates the qualifications which expire on the date shown, or which have already lapsed [7]. If there are none, test [8] jumps directly to [12].

On the other hand, if certain qualifications must be renewed, we calculate those which have lapsed in *TOOLATE*. Line [10] prints all qualifications to be renewed. Line [11] selects, in the matrix *SIGNS*, either a series of blanks, or an underlining dash, in terms of *TOOLATE*. The dimensions of *SIGNS* conform with the format used to print the qualifications.

THE PLOT THICKENS

Certainly not! When seeking a qualification, it is sufficient to replace it by the list of qualifications which cover it.
For example, when seeking welders who have qualification 308, we must in fact search for those which possess any one of qualifications 308, 313, 500, 510, 312.

It is sufficient to modify the right-hand argument of *WELD*1 as follows:

```
[0.5] LIST ← LIST, (∨/EQUIV∊LIST)/QUALIF
```

The same modification applies to *WELD*2, but the matrix *MAT* may contain several non-zero dates for the same welder.
Now take the case where we are looking for welders who have renewed their 409 since 1980.

- the list of qualifications concerned is: 409 and 512
- the welder *MATTHEWS* possesses both these qualifications: the corresponding line of *MAT* will thus equal:

  82 81 0 0 0

- if, as previously, we add these, we will find 163!

Hence, the most recent date must be obtained by reducing by the maximum.

```
      ∇ QUAL WELD2B DATE ;MAT;CACHE;LIST
[1]     LIST←QUAL, (∨/EQUIV∊QUAL)/QUALIF
[2]     MAT←WELDATE × WELQUAL∊LIST
[3]     CACHE←∨/MAT>DATE
[4]     (CACHE/[1]WELDERS), 4 VERT CACHE/[1] ⌈/MAT
      ∇
```

FOR AESTHETES ONLY

We can resume the case of *WELD*1. The *CACHE* enables the names to be printed; it must also allow the columns of numbers to be printed.

In order to join the two matrices *WELQUAL* and *WELDATE* and a single matrix which has 10 columns, we will use one of the three possible forms of the function *IMBRICATES*, written in the "*Hope and Truth*" topic.

Hence we obtain, in line [2], a matrix called *COUPLES* which contains alternately a column of qualifications and a column of dates.

We can then write the function:

```
      ∇ WELD1C LIST ;CACHE;COUPLES;NAMES
[1]     CACHE← ∨/WELQUAL∊LIST
[2]     COUPLES← CACHE/[1] WELQUAL IMBRICATE WELDATE
[3]     NAMES ← CACHE/[1] WELDERS
[4]     NAMES, (20ρ 6 0 3 0) FORBLANK COUPLES
      ∇
```

On line [4] we could have written *NAMES*, (20ρ 6 0 3 0) ⍕ *COUPLES* but the zeros would have been printed.
We have therefore replaced the function FORMAT by a defined function called *FORBLANK*, whose syntax is the same, but which eliminates the non-zero values.
Here it is:

```
      ∇ R← FOR FORBLANK ARR ;DIM;POS;DIM
[1]     DIM←ρR←FOR ⍕ ARR
[2]     R←,R
[3]     POS←(R='0') ∧ 1, ¯1↓R=' '
[4]     R[POS/⍳ρPOS]←' '
[5]     R←DIMρR
      ∇
```

We start by using the FORMAT function normally, and we note in *DIM* the dimensions of the result obtained [1]. Working on a vector, however, is easier, and we ravel the matrix obtained [2].

The insignificant zeros are those which are preceded by a blank.

- *R*='0' gives the position of the zeros (note: in characters)
- *R*=' ' gives the position of the blanks, which we shift by ¯1↓*R*=' '

For cases where the first character of a line would be zero, we work, as though it were preceded by a blank, by means of a 1 catenated at the beginning.

We calculate the indices of these insignificant zeros and replace them by blanks [4].
It remains only to re-establish the result in its instruction [5].

Of course, this function works only in the present context, when the printing format contains no decimals. For formats including decimals, there is quite a different solution.

WILL IT BE FINE TOMORROW?

There are actually not two but four solutions available to us for solving this problem, according to the significance we attach to the letters *I*,*L* and *A*, typed by the user.

If the introduction of data is undertaken by a QUAD:

- *I*, *L*, *A* can be the names of local variables,
- these can be names of niladic functions which give a result.

If the introduction of data is undertaken by means of a QUOTE-QUAD:

- *I*, *L*, *A* can designate function names, as in the previous case,
- unless we prefer to proceed with a test and jumps.

We will examine these four possibilities.

FIRST SOLUTION WITH A QUAD

```
      ∇ R←TEXT EXTRAPOLATE ARR ;LINE;I;A;L;NB
[1]     NB←1↑⍴R←ARR
[2]     LINE←1
[3]     AGAIN:M←⌊AVERAGE I←ARR[LINE;]
[4]     D←¯1↑I
[5]     TEXT[LINE;]
[6]     R[LINE;]←⎕
[7]     →(NB≥LINE←LINE+1)/AGAIN
      ∇
```

A loop enables all the lines of an array to be scanned in sequence, in all the solutions.
The line index (*LINE*) leaves the value 1 (instruction [2]), to reach the value *NB* (instruction [7]), *NB* being the number of lines of the given array [1].

For each line processed, we display the corresponding label by instruction [5], then in [6] we collect the values typed by the user.

It is here that the solutions differ.

In the solution shown here, *I* is a local variable which contains the appropriate line of the array supplied on input. From the function *AVERAGE* we calculate the average *M* of these values [3].
Finally, *D* contains the last value of line [4].

When the user types *I* or *A* or *L*, the VALUE of the local variable *I*, *A* or *L*, is received by the QUAD.

SECOND SOLUTION WITH QUAD

In order to avoid systematically calculating three variables *I*, *A* and *L*, when there is a chance that only one will serve, we can define these words as function names.
These functions will be executed by the QUAD and will thus provide for reference to the variables created by *EXTRAPOLATE*, which will be global to them.

```
      ∇ R←I
[1]   R←ARR[LINE;]
      ∇

      ∇ R←M
[1]   R←⌊AVERAGE ARR[LINE;]
      ∇

      ∇ R←D
[1]   R←¯1↑ARR[LINE;]
      ∇

      ∇ R←TEXT EXTRAPOLATE ARR ;LINE;NB
[1]   NB←1↑ρR←ARR
[2]   LINE←1
[3]   AGAIN: TEXT[LINE;]
[4]   R[LINE;]←⎕
[5]   →(NB≥LINE←LINE+1)/AGAIN
      ∇
```

The principal program is more concise, since it receives suitable prepared values directly by its QUAD.

This solution has the shortcoming of requiring auxiliary functions, but it generalises itself immediately to the QUOTE-QUAD.

FIRST SOLUTION WITH QUOTE-QUAD

We will retain the auxiliary functions *I*, *L* and *A* created for the last solution.

Replacing the simple QUAD by a QUOTE-QUAD, the user's answer is no longer evaluated, but gathered like a character string.
The EXECUTE function provides for calculation of the value representing this string. This remains valid in the case where the user has typed a series of numbers, where he has typed *I*, *L* or *A*.

```
      ∇ R←TEXT EXTRAPOLATE ARR ;LINE;NB
      NB←1↑⍴R←ARR
      LINE←1
      AGAIN: ⍞←20↑TEXT[LINE;]
      R[LINE;]←⍎⍞
      →(NB≥LINE←LINE+1)/AGAIN
      ∇
```

The relationship with the previous solution is striking. We have simply replaced ⎕ by ⍎⍞.
Note also the use of "bare-output", which enables a question to be formulated and answered on the same line.

SECOND SOLUTION WITH QUOTE-QUAD

If we wish to avoid recourse to auxiliary functions, we must proceed by a discriminatory test between the various answers possible. The result is infinitely more tedious.

```
      ∇ R←TEXT EXTRAPOLATE ARR ;LINE;NB;ANSWER
      NB←1↑⍴R←ARR
      LINE←0
      AGAIN: →(NB<LINE←LINE+1)/0
      ⍞←20↑TEXT[LINE;]
      ANSWER←20↓⍞
      →('IAL'∊ANSWER)/AGAIN,AV,LAST
      R[LINE;]←⍎ANSWER
      →AGAIN
      AV: R[LINE;]←⌊AVERAGE ARR[LINE;]
      →AGAIN
      LAST: R[LINE;]←¯1↑ARR[LINE;]
      →AGAIN
      ∇
```

If none of the conventional letters features in the answer, we place the value typed by the user in the line of the result, after evaluating it by means of ⍎.

If the answer contains one of the letters *A* or *L*, we move to the instructions whose task it is to treat these cases: [9-10] for the average [11-12] for the last value.

As the result *R* takes the value of *ARR* at the beginning of the processing, it is pointless to modify it if the user's answer is *I*. This explains the direct return to *AGAIN*.

Control of the loop has consequently been modified.

CRITICAL STUDY

Taking again the terms of the problem, it appears that the solutions which summon the QUOTE-QUAD offer more agreeable presentation than those using the simple QUAD.
It also consumes less paper, and is more ecological.

Another hazard: we can type any expression on a QUAD and it is its value which will be collected. Bad luck to the absent-minded person who inadvertently types the name of a variable other than *I A L*, whose contents are incompatible with the processings carried out.

Quad supporters reply that it is also an advantage, since one can also answer by expressions such as:

1.2×*I* or 2+*L*

The disadvantages of the QUOTE-QUAD are similar. Since the QUOTE-QUAD has to be followed by an EXECUTE, it is necessary to check the contents of the character string to be executed. In particular, it must not be empty.

If the values introduced must be whole, the checks are fairly easy, because the characters of *ANSWER* must be limited to the following alphabet:
'0123456789 *IAL*'

If however, the values can be decimal, the checks must be multiplied, so as to eliminate typing errors such as:

345..60 or .¯44 or 77¯46 or 12.34.56

Unless, of course, you are the fortunate user of an APL system offering a control function as standard which carried out these modifications. It is extremely useful!

CROSSED NUMBERING

FIRST STEP

```
      ∇ R←CROSS MAT
[1]   R←MAT[;1] CRUC MAT[;2]
      ∇
```

The function *CROSS* receives a matrix as an argument. It separates it into two vectors and executes calculations by the function *CRUC*, which will serve us later on.

```
      ∇ R←V CRUC H ;VER;HOR;BINH;BINV
[1]   VER←CONDENSE V
[2]   HOR←CONDENSE H
[3]   BINV←VER∘.=V
[4]   BINH←H∘.=HOR
[5]   R←TOTALISER BINV+.∧BINH
      ∇
```

In [1] we look for the list of all the distinct values which on the left hand argument can take. This task is transmitted to the function *CONDENSE*. The same work is carried out in line [2] for the other argument.

In the case of the example chosen:

- *VER* would be the vector 0 1
- *HOR* would be the vector 420 452 471 519 833 953

Hence two binary matrices are constituted which indicate, by the 1's, the commercial engineer responsible for each contract, and the corresponding type of client (instructions [3 and 4]).
With the calculation performed, *BINV* has 2 lines and *n* columns, whereas *BINH* has *n* lines and 6 columns.

We are concerned now with the <u>TOTAL</u> contracts relating to a certain clientele <u>AND</u> with a certain commercial engineer. This indicates an inner product by +.∧, which will give a result of 2 lines and 6 columns. This is classic APL.

This product was used to solve exercise 33, page 159 of the course book. The simple matricial product +.× would be equally suitable, but +.∧ is closer to the expression of the problem in French.

The *TOTALISER* function serves only to border the result by its lines and columns totals:

```
      ∇ R←TOTALISER X
[1]   R←X,[1]+/[1] X←X,+/X
      ∇
```

Return to the *CONDENSE* function

The first instruction serves to eliminate the values which appear several times in the argument, so as to keep the list of very distinct values only. This classic method is studied in the coursebook, page 392.

The second line enables these values to be classed in increasing order, to facilitate reading of the results.

```
      ∇ R←CONDENSE X
[1]   R←((X⍳X)=⍳⍴X)/X
[2]   R←R[⍋R]
      ∇
```

SECOND STEP

In order to write the references, the values *VER* and *HOR* must be known by the *CROSS* function. We have also modified the header of *CRUC* and *CROSS* so that these variables are GLOBAL for *CRUC* and LOCAL for *CROSS*.

```
      ∇ R←V CRUC H ;BINH;BINV
```

The variable *FOR*, giving the width of the columns of figures, the horizontal reference *REFH* written at the top of the array comprises:

- a blank zone	*FOR* ⍴ ' '
- the values of *HOR*, well formatted	(*FOR*,0) ⍕ *HOR*
- the entry "TOTALS"	(-*FOR*) ↑ '*TOTALS*'

The vertical reference *REFV* comprises the values of *VER*, placed in column form by means of the utilitarian function *VERT*, then the entry "TOTALS".

The complete array is constituted in line [4].

```
      ∇ R←CROSS MAT ;HOR;VER;REFH;REFV
[1]   R←MAT[;1] CRUC MAT [;2]
[2]   REFV←(FOR⍴' '),((FOR,0)⍕HOR),(-FOR)↑'TOTALS'
[3]   REFH←(FOR VERT VER),[1](-FOR)↑'TOTALS'
[4]   R←REFH,[1]REFV,(FOR,0)⍕R
      ∇
```

This array is emitted as the explicit result of the function, which is rarely the case concerning array print-outs. This option offers the advantage of enabling the user to use this result in a complementary function which could frame it, write it on a file, etc...

THIRD STEP

If we know the matrices *BINH* and *BINV*, we can multiply one or other by the total of all contracts. For reasons of dimensions, the easiest to multiply is *BINV* (instruction [2]).

Just as the product *BINV*+.∧*BINH* gave a sum of 1 and 0, the product *P*+.×*BINH* gives a sum of amounts. This is the result sought.

The economical method for obtaining *BINH* and *BINV* would consist of again using lines [1] to [4] of *CRUC*. Here we have preferred to call upon the *CRUC* function itself [1], even though its result (*NUMBERS*) is of no use to us at the moment.

```
      ∇ R← P BALANCE MAT ;NUMBERS;BINH;BINV
[1]     NUMBERS←MAT[;1] CRUC MAT[;2]
[2]     P←BINV × (ρBINV)ρP
[3]     R←TOTALISER P+.×BINH
      ∇
```

Note, that for this, *BINV* and *BINH* had to be made global for *CRUC*. They must thus be made local for *BALANCE*, and for *CROSS* also.

LAST STEP

The function is composed of a series of calculations [1 to 6], then a suitable presentation of the results [7 to 12].

CALCULATIONS

We will need the number of contracts of each sort; the simplest is to summon *CRUC* as in the preceding step [1].

The turn-over achieved by each commercial engineer is calculated as in the preceding step [2 and 3].

We can constitute an array of three dimensions whose three "planes" contain respectively, for each case indexed, the total turn-over (like step 3), the number of contracts (like step 1), the average *R*÷*NUMBER*. The entry *R*÷*NUMBERS*⌈1 avoids divisions by zero.
Lamination is the easiest way of achieving this. The result *R* has dimensions 3 2 6.

However, we want an array of 6 columns comprising 2 groups of 3 lines, or 6 lines, or ×/2↑ρ*R* lines. To fill it, we must take a "turn-over" line, a "number of contracts" line and an "average" line.
This is not the order in which the information contained in *R* is actually presented.

R being of dimensions 3 2 6, we will return it to dimensions 2 3 6 by the re-ventilation 2 1 3⍉*R* which exchanges the first two dimensions.
Such is the meaning of instruction [5].
It is thus possible to constitute the final array [6].

```
       ∇ R←P BALANCE MAT ;NUMBERS;BINH;BINV;REFH;REFV;TOP;HOR;VER
[1]    NUMBERS←MAT[;1] CRUC MAT[;2]
[2]    P←BINV × (⍴BINV)⍴P
[3]    R←TOTALISER P+.×BINH
[4]    R←R,[1] NUMBERS,[0.5] R÷NUMBERS⌈1
[5]    R← 2 1 3 ⍉R
[6]    R←((×/2↑⍴R),¯1↑⍴R)⍴R
[7]    REFH←(FOR⍴' '),((FOR,0)⍕HOR),(-FOR)↑'TOTALS'
[8]    REFV←(FOR VERT VER),[1](-FOR)↑'TOTALS'
[9]    TOP←4× 1↑⍴REFV
[10]   REFV←(TOP⍴ 0 1 0 0)\[1] REFV
[11]   R←(TOP⍴ 0 1 1 1)\[1](FOR,0)⍕R
[12]   R←REFH,[1] REFV,R
       ∇
```

PRESENTATION

Instructions [7 and 8] are similar to those of step 2, but the presentation must be expanded in the vertical direction. The final number of lines, *TOP*, is equal to 4 times 1↑⍴*REFV*.

In [10] and [11], two binary expansion vectors clearly show that we have:

- 1 reference line for 4 result lines : 0 1 0 0
- 3 figure lines for 4 result lines : 0 1 1 1

It suffices to join the pieces together again in line [12].

The expansion would suggest calling into use the *PRINT* function, written in the topic "*Everything is in the presentation*". For this it would suffice to add a thirteenth instruction in order to calculate the position of the lines and columns, and a final one to undertake the layout:

```
[13]  POS←(1+⍴VER), 1 4 , (1+⍴HOR), FOR+ 1 0
[14]  R ← POS PRINT R
```

A DIFFICULT CHOICE

In the following solutions, we have assumed that the codes supplied were suitably constituted of one letter followed by three figures. If this was not the case, it would be advisable to undertake the necessary controls before any other treatment no matter what the solution for the search.

FIRST STRUCTURE

We start by searching for all the codes whose numeric part is identical to that of the code sought. We could extract the latter by:

```
⍎ 1↓X
```

Where we want to be covered by a maximum precaution, it would be preferable to write:

```
1↑ ⍎ ((X∊'0123456789')/X),' 0'
```

The compression (X∊'0123456789')/X enables only the figures to be kept, even if they are not at the head. But it could be that no figure typed, and applied to an empty vector, would give a value error. A good safeguard hence consists of catenating a blank and zeros to the character string obtained. Application of the function ⍎ will give either a scaler or a vector, of which we will retain only the first term, by 1↑.
It is seen that prevention of possible errors is rather tedious.

In the function below, we have retained the simple solution, will all the risks which it entails.

```
      ∇ R←FIND1 X
[1]     R←(PRONUM=⍎1↓X)/⍳⍴PRONUM
[2]     R←1↑(PRORAN[R]=1↑X)/R
      ∇
```

The first instruction gives the indices of the numeric codes identical to the one we are looking for.
In the second line, *PRORAN*[*R*] gives the corresponding letters, and we compare them to the letter of the code sought: 1↑*X*. Compression however, gives either an index, or an empty vector, if the code sought does not exist. This is why we have performed 1↑... , so as to transform a possible empty vector into zero.

SECOND STRUCTURE

Looking for a vector of characters in a matrix is indeed a classical problem, studied in detail in the course, page 120. Hence we will write:

PROMAT ∧.= *X*   or, better still *PROMAT* ∧.= 4↑*X*

The result obtained is a binary vector comprising a single 1, if the code exists, or composed uniquely of zeros if it does not exist.
In order to know the position of the code sought, we can proceed with compression followed by 1↑, as in the preceding solution:

```
      ∇ R←FIND2 X
[1]   R←PROMAT ∧.= 4↑X
[2]   R←1↑ R/ιρR
      ∇
```

We can also look for the position of the 1 by the INDEX-OF function, which must be quicker if the code sought is situated in the initial ones. If however it does not exist, we will obtain the number of codes plus 1 as result, whence multiplication by *R*≤1↑ρ*PROMAT* to transform this possible value to zero:

```
      ∇ R←FIND2B X
[1]   R←(PROMAT∧.= 4↑X)ι1
[2]   R←R × R≤1↑ρPROMAT
      ∇
```

It will be seen, during the trials, that the gain in time is very small.

This solution is much sounder than the preceding one, because it does not produce an error if the code introduced is wrongly constituted.

THIRD STRUCTURE

If we subject the code sought to the same transformation to which we subjected the *PROMAT* matrix, we obtain a numeric value. It therefore suffices to look for its position in *PROVEC* by the INDEX-OF function. As in the preceding case, a second line transforms the result to zero if necessary.

```
      ∇ R←FIND3 X
[1]   R←PROVEC ι 36⊥ALPHANUMιX
[2]   R←R × R≤ρPROVEC
      ∇
```

This solution presents certain dangers. First of all it would be wise to work in 4↑*X*. Now the presence of an abnormal character in the code sought will

result in the appearance of the value 37 when *ALPHANUM*⍳*X* is undertaken. Decoding in base 36 will give a value which may being identical to an existing code.

Likewise *A**46 would be confused with *B*046. This is rather tedious.

FOURTH STRUCTURE

```
     ∇ R←FIND4 X
[1]    R←(⍎1↓X) + 1000×ALPHA⍳1↑X
[2]    R←(R≤⍴PROVECB) × R←PROVECB⍳R
     ∇
```

To use this variant, we have consigned the vector of the 26 letters of the alphabet in *ALPHA*, and we have transformed the list of codes in the following way:

```
PROVECB ← PRONUM + 1000×ALPHA⍳PRORAN
```

It thus suffices to cut the code sought into its numeric part and its letter part in order to apply the same transformation to it.

This method summates the various risks: those associated with the presence of the EXECUTE function, and those mentioned for the preceding structure.

FIFTH STEP

```
     ∇ CALTIME NB ;TIME;USELESS;I;EXP
[1]    IB: ⍞←20↑'EXPRESSION :'
[2]    →(0=⍴EXP←20↓⍞ )/0
[3]    I←1
[4]    TIME←⎕AI[2]
[5]    REPEAT: USELESS← ⍎ EXP
[6]    →(NB≥I←I+1)/REPEAT
[7]    'TIME = ',⍕⎕AI[2]-TIME
[8]    →IB
     ∇
```

The expression studied is introduced in the form of a character string *EXP*, by means of instructions [1-2]. We can note in *TIME* the central unit time consumed before execution.

A loop then enables the given expression to be executed *NB* times. It remains only to compare the time consumed before and after execution.

To avoid printing the results rendered by execution of *EXP*, we have assigned the latter to a variable *USELESS*. This technique is dispensed with if we wish to introduce an expression which does not give a result.

LAST METHOD

If the operand is introduced in numeric form, all the problems mentioned vanish and the function is greatly simplified:

```
      ∇ R←FIND6 X
[1]     R←PROVECB ⍳ X
[2]     R←R×R≤⍴PROVECB
      ∇
```

MEASUREMENTS RESULTS

The time measured can vary significantly according to the computer used, the APL version introduced and the detail of the functions written. Certain solutions are sensitive to the position of the code sought in the list of codes, others are less so.

Here are the times observed for the ten executions of the aforementioned functions:

FIND1	155 milliseconds
FIND2	220 milliseconds
FIND2B	218 milliseconds
FIND3	70 milliseconds
FIND4	80 to 95 milliseconds
FIND6	50 to 65 milliseconds

As anticipated, apurely numeric search is quicker than any other solution. If however we wish to retain the codes in their alphanumeric form, solutions using decoding are the fastest, despite the apparent complexity of this work for man.

This exercise clearly demonstrates the richness of the solutions offered by APL. We maintain that the use of decoding is an excellent method for copying with alphanumeric code searching problems.

BLOCK AND TACKLE

In a preparatory phase, we start by:

[2] defining a set of characters serving to plot the blocks. Here we stop at 10 characters, which will be re-used in rotation if we have to plot more than 10 blocks per group (which is rare),

[3] calculating the height of the graph, which is that of the highest block,

[4] and noting in *RHO* dimensions of the given matrix,

[5] filling a matrix which has the dimensions of the final graph with blanks. This matrix will be filled by the suitable characters.
Calculation of the dimensions as follows:

> There are *RHO*[2] groups, whose width is equal to: 4×*RHO*[1]-1 for the first blocks of the group, + 6 for the last, + 3 for the blank space which follows, i.e., after a simple calculation, *RHO*[2]×5+4×*RHO*[1]

We then fill the matrix *R* block by block. *LI* and *CO* are the indices of the line and column of *MAT* during processing. For each value of *LI* and *CO* we plot a block whose height is *MAT*[*LI*;*CO*]. The character used could be *CAR*[*LI*]. We prefer *CAR*[1+10|*LI*-1] so as to use the 10 characters defined in rotation.

Each block takes its place in *R* in the *INDLI* lines and in the *INDCO* columns. At the beginning, *INDCO* equal ⍳6 (instruction [7]), then as we pass form one block to another [13], the index increases by 4. Hence each block overlaps the preceding block plotted.

On the other hand, when we pass from one group of blocks to the next [16], *INDCO* increases by 4 (in the line [13]) and again by 5 (in line [15]), so that the groups are separated.

The indices of the lines which a block must occupy are calculated in [11]. These are indices which start at the bottom of the matrix *R* and are of the height of the block, i.e 1↑⍴*BLOCK*

Two loops [9 and 16] and [10 and 14] cause *CO* and *LI* to vary by 1 to *RHO*[2] or *RHO*[1].

The function is shown in detail on the following page.

```
     ∇ R←MULTIBLO MAT ;CAR;TOP;RHO;INDCO;CO;LI;INDLI;BLOCK;ECHY
[1]    ⍝                    PREPARATION OF DATA
[2]    CAR ← '\⎕⋆|○/-∇×∆'
[3]    TOP ← ⌈/,MAT
[4]    RHO ← ⍴MAT
[5]    R ← (TOP , ¯3+RHO[2]×5+4×RHO[1])⍴' '
[6]    ⍝                    WRITING THE BLOCKS IN R
[7]    INDICO ← ⍳6
[8]    CO ← 1
[9]    NEXTCO: LI ← 1
[10]   NEXTLI: BLOCK ← (MAT[LI;CO],6)⍴CAR[1+10|LI-1]
[11]   INDLI ← 1+TOP-⍳1↑⍴BLOCK
[12]   R[INDLI;INDCO] ← BLOCK
[13]   INDCO ← INDCO+4
[14]   →(RHO[1]≥LI←LI+1)/NEXTLI
[15]   INDCO ← INDCO+5
[16]   →(RHO[2]≥CO←CO+1)/NEXTCO
[17]   ⍝                    WRITING THE SCALES
[18]   R ← (((TOP,2)⍴' '),R),[1]'-'
[19]   ECHY←(3 VERT ⌽⍳TOP),' ',[1.5] ⌽TOP⍴'⊦|||-'
[20]   R ← (ECHY,[1]'----|'),R
     ∇
```

The last step consists of placing the scales. Two blank columns border *R* on the left and a horizontal dotted line is plotted below [18]

We have used the *VERT* function to constitute the vertical scale by means of ⍳*TOP* reversed.

The final result is emitted as the function result. This is not a widespread practice in graph plotting but it offers the advantage of returning a characters array to the user which he can centre, or supplement with additional references.

IN THE TIME OF THE PYRAMIDS

PREAMBLE

All the dates must be expressed in a common unit. The number of months elapsed from a fixed origin is established.
Working for example, on recruitment dates, three methods yield the same result:

a) *ES*[;1] + 12 × *ES*[;2]

b) *ES*[;1 2] +.× 1 12 which amounts to the same thing

c) 1 12 ⊥ ⍉ *ES*[;1 2]

The third solution is decoding, similar to the expression which enables an hour to be converted into seconds: 24 60 60 ⊥ *HOUR*

The second and the third methods are 2 and 3 times slower than the first respectively. It is therefore this one which we will retain.

The function *PREPARE* will carry out these preliminary calculations, and will leave two global variables in the workspace: *RECRUIT* and *LEAVE*:

```
      ∇ PREPARE
[1]   RECRUIT ← ES[;1] + 12×ES[;2]
[2]   LEAVE ← ES[;3] + 12×ES[;4]
      ∇
```

FIRST STEP

If we undertake a check at a given date, it is advisable to express it in months for the same reason as the others [1].

The people answering two conditions are present on this date:

- recruitment before the given date,
- AND departure after the given date. In the expression of the latter condition, people still present must not be forgotten (*LEAVE*=0)

```
      ∇ R←CUT DATE ;PRESENT
[1]   DATE ← DATE[1] + 12×DATE[2]
[2]   PRESENT ← (RECRIUT≤DATE)∧(LEAVE>DATE)∨(LEAVE=0)
[3]   R ← ⌊ (DATE-PRESENT/RECRUIT)÷12
[4]   R ← +/SCOPE ∘.= R
[5]   R ← R,+/R
      ∇
```

The length of service on months, is the difference between the check date and the recruitment date, only for those present, of course. But to express it in whole years, we must devide by 12 and round down [3].

Hence *R* is the vector of the years of service of all the employees present at the check date.

We have assumed, in [4], that the global variable *SCOPE* contained the scope of variation observed. In the present case: the whole numbers from 0 to 20. Distribution by groups is thus obtained by a classical outer product ∘.=

It remains only to catenate *R* with its total [5].

SECOND STEP

As stipulated, we start by converting the data [1]. Hence it suffices to execute *CUT* by means of a loop for the month of December of the *n* prescribed years. We thus obtain *n* vectors, which we can:

- either join one under another into a matrix by catenation,
- or arrange in the *n* lines of a pre-defined matrix, the solution which has been retained here, in line [4]. Since, the scope of the work is written in the argument, the matrix will have the dimensions:

  ρ*DATES* lines (one per year),

  1+ρ*SCOPE* columns (because of the cumulation per year)

```
      ∇ SCOPE PYRAMIDS DATES ;RECRUIT;LEAVE;I;R
[1]     PREPARE
[2]     R ← ((ρDATES),1+ρSCOPE)ρ0
[3]     I←1
[4]     LO: R[I;] ← CUT 12,DATES[I]
[5]     →((ρDATES)≥I←I+1)/LO
[6]     ''
[7]     (4ρ' '), 4 0⍕SCOPE
[8]     ''
[9]     4 0 ⍕ DATES,R
      ∇
```

Instructions [6 to 9] are included to give a suitable presentation to the result.

FLASH

FIRST PART

We define an array by its header and the rest of the lines to be printed.

1 - The header is collected by instructions [1 and 2] based on the "bare out-put" technique (see course page 107).
A test serves to check that this header does not exceed 30 characters. Otherwise a message is printed and we start again [3].

2 - The line numbers are collected by instructions [4] and [5]. Instructions [6] and [7] serve to check that the codes introduced indeed feature in the vector *CODES* (line [6]). In the event of error, abnormal codes are printed [7]. We check also that not more than 30 values have been introduced [8 and 9].

If those conditions are fulfilled, the information collected is calibrated to 30 elements by ↑ and catenated to *HEADREST* and *CODREST* respectively. Hence we display the number of the defined array.

We could have provided a loop in order to define several arrays in series.

```
      ∇ DEFREST ;H;LI;BIN;DIM
[1]   LO: ⍞←'ARRAY HEADER ....'
[2]   →(30≥ρT←20↓⍞ )/L1
[3]   →LO AFTER 'TOO LONG'
[4]   L1: 'LINES TO BE PRINTED'
[5]   L1←,⎕
[6]   →(∧/BIN←LIε0,CODES)/L2
[7]   →L1 AFTER 'ABNORMAL CODES : ',⍕(~BIN)/LI
[8]    L2: →(30≥ρLI)/L3
[9]   →L1 AFTER '30 LINES MAXIMUM PLEASE'
[10]  L3: CODREST←CODREST,[1] 30↑L1
[11]  DIM←1↑ρHEADREST←HEADREST,[1] 30↑T
[12]  'THIS ARRAY WILL BEAR THE NUMBER ',⍕DIM
      ∇
```

The function uses the slave function *AFTER* to print the error messages. Note, in line [5], the ravelling of the values entered on the keyboard. If we do not take this precaution, testing line [8] would not work, in the case where we were to introduce only one number (scalar).

SECOND PART

The list of defined arrays is obtained by simply displaying *HEADREST*, bordered by a series of whole numbers. The function *VERT* enables these numbers to be arranged in a column:

```
      ∇ ARRAYS ;NUM
[1]   NUM←2 VERT ι1↑ρHEADREST
[2]   NUM,((ρNUM)ρ' '),HEADREST
      ∇
```

THIRD PART

The number of the array to be printed is given as argument; in [1 and 2] we check that it is there.
Then, having extracted the corresponding line of *CODREST*, the zeros on the right have to be eliminated. Now, we know how to eliminate zeros on the left by means of a scan (see course, bottom of page 113). Hence, we will tackle this familiar problem (don't smile!) by a double reversal [3-4].
After this, only the codes of the useful lines are left in *LI*, separated by zeros

```
      ∇ PRIREST NUM ;LI;POS;BIN;R
[1]   →(NUM∈ ι1↑ρHEADREST)/LO
[2]   →0 AFTER 'ABNORMAL ARRAY NUMBER'
[3]   LO: LI←⌽CODREST[NUM;]
[4]   LI←⌽(∨\0≠LI)/LI
[5]   LI←(BIN←LI≠0)/LI
[6]   POS←CODESιLI
[7]   ''
[8]   HEADREST[NUM;], 13 0 3 0 5 0 ⍕⎕TS [3 2 1]
[9]   51ρ'-'
[10]  ''
[11]  R←(2 VERT LI),' ',(POSTS[POS;]), 6 0 ⍕ RESULTS[POS;]
[12]  BIN\[1]R
      ∇
```

In [5] we eliminate these zeros so as to look for the indices of the lines to be printed in [6] (*POS*).
This set of indices serves to print side by side, in [11]:

- the codes, placed vertically by the utilisation function *VERT*,
- the associated labels: *POSTS[POS;]*
- the corresponding results.

The binary vector *BIN* calculated in [5] provides for the expansion necessary for insertion of blank lines to be undertaken at the positions of the zeros in the given model.

Lines [8 to 10] serve to present the header.

This is actually very simple; make sure you use it for all your restitution problems.

WEEK-END SAILING AT BRIGHTON

DATA STRUCTURE; INITIALISING

The information necessary for this work is already partly known:

- *OUTNAMES* matrix of outings names, with 30 columns,
- *OUTPRICES* vector of participating prices for these outings,
- *OUTCODES* vector of numbers assigned to the outings,
- *PERREGS* vector of employees registrations,
- *PERNAMES* matrix of their names (12 columns),

We then complete the 5 following vectors for each registration:

- *INSREG* vector of employees registered, in the order of registration,
- *INSOUT* numbers of the outings requested by each one,
- *INSNB* number of people registered,
- *INSCASH* total instalments paid,
- *INSOWE* total still owing.

For example:

INSREG	115	241	122	245	115	122	122	etc ...
INSOUT	6	6	7	7	8	6	8	
INSNB	2	1	3	2	1	1	3	
INSCASH	1000	200	1500	500	600	750	500	
INSOWE	500	550	0	500	0	0	1300	

It can be clearly seen that the same employee appears as many times as he has requested outings. These vectors contain in another form, the information which the function *STATE* will print.
A simple function will serve to initialise all these variables, except those concerning the personnel:

```
      ∇ INIOUT
[1]     OUTNAMES←0 30⍴''
[2]     OUTPRICES←OUTCODES←INSREG←INSOUT←INSNB←INSCASH←INSOWE←⍳0
[3]     END←*1
      ∇
```

The global variable *END* will be used for tests to bring out the functions.

ORGANISING AN OUTING

This is a very simple function. We introduce the outing description and the individual participation price successively. These two pieces of information are catenated to the appropriate data.

Since the vector *OUTCODES* contains the numbers in increasing order, the number attributed to the last outing created is obtained by ¯1↑*OUTCODES*. It is better to avoid ⌈/*OUTCODES* because reduction by the maximum of an empty vector gives an enormous number. This exceptional case will arise on entering the first outing, since *OUTCODES* has been initialised by an empty vector.

```
      ∇ PLAN ;NUM
[1]   'OUTING DESCRIPTION ?'
[2]   OUTNAMES←OUTNAMES,[1]30↑⍞
[3]   'INDIVIDUAL DONATION ?'
[4]   OUTPRICES←OUTPRICES,⎕
[5]   NUM←1+ ¯1↑OUTCODES
[6]   '--- THIS OUTING WILL BEAR CODE', ⍕NUM
[7]   OUTCODES←OUTCODES,NUM
[8]   SAVEWS
      ∇
```

At the end of the work, the funciton *SAVEWS* serves automatically to safeguard the workspace.
Achieving such a function depends greatly on the APL system used. On other systems, it suffices to write ⎕*SAVE* ; on others one can write ⍎')*SAVE*';
on VS-APL systems offered by IBM, one must use the STACK auxiliary processor (see course, page 309), as follows:

```
      ∇ SAVEWS; STACK
[1]   →(2≠101 ⎕SVO 'STACK')/AN
[2]   STACK←')SAVE'
[3]   →0
[4]   AN: '101 PROCESSOR IN THE ROADS'
      ∇
```

Of course this step is not essential to the working of programs, but it provides automatic safeguarding of the information entered. We would obtain an equivalent level of security by working on a file.

REGISTRATIONS AND INSTALMENTS

For registrations input, two loops are imbricated. They enable several outings to be surveyed, and several employees to be registered for each one.

First of all we introduce the outing code [1 to 3]; if we type *END* the program finishes by a safeguard [15]. If not, we look for the index *POS* of this outing in the list of outings. If *POS* is greater than the outing number, it is because the code *OUT* is incorrect and the jump [5] returns to the question:

```
      ∇ INSCRIPTIONS ;OUT;POS;REP;NB;TO;FI;REG;LO
[1]   LO:  ''
[2]   'OUTING CODE'
[3]   →(END=OUT←⎕ )/LEAVE
[4]   POS←OUTCODES⍳OUT
[5]   →(POS>⍴OUTCODES)/L0
[6]   L1: 'REGISTRATION, NUMBER OF PARTICIPANTS'
[7]   NB←(REP←2↑⎕ )[2]
[8]   →(END=MAT←REP[1])/L0
[9]   →(~MAT ∊ PERREGS)/L1
[10]  'DONATION : ',⍕ MD←NB×OUTPRICES[POS]
[11]  'FIRST INSTALMENT'
[12]  PV←⎕
[13]  CONSERVE
[14]  →L1
[15]  LEAVE: SAVEWS
      ∇
```

In [6 to 9] we enter the registration and number of participants into *REP*. *NB* will receive the number of participants and *REG* will receive the employees registration.
When we type *END*, it is to conclude the registration for this outing. The jump [8] thus moves to the next outing.

In [10] we display the participation price, which is the product of the number of participants and the price of the outing requested. Thus we can enquire what the employees first instalment will be and proceed to the following employee by means of the jump [14].

The slave function *CONSERVE* serves to ventilate the information induced in the various global variables.

If we wish to display the outing name, as in the example, we must insert an instruction:

```
[5.5] OUTNAMES[POS;]
```

```
     ∇ CONSERVE
[1]  INSREG   ←  INSREG,REG
[2]  INSOUT   ←  INSOUT,OUT
[3]  INSNB    ←  INSNB,NB
[4]  INSCASH  ←  INSCASH,FI
[5]  INSOWE   ←  INSOWE,TO-FI
```

This is not an intellectualy fantastic function, but it is useful.

For input of further instalments, a loop enables pairs of registration-outing numbers [1 to 3] to be entered. The value *END*, when detected in [3], proceeds to the safeguard instruction in [13].

```
     ∇ INSTALMENT ;OUT;CACHE;POS;I;REP
[1]  LO: ''
[2]  'REGISTRATION, OUTING'
[3]  →(END ∊ REP←2↑⎕) /LEAVE
[4]  CACHE←INSREG=REP[1]
[5]  OUT←REP[2]
[6]  →(~OUT CACHE/INSOUT)/LO
[7]  POS←((OUT=INSOUT)∧CACHE)/⍳⍴INSREG
[8]  'ALREADY PAID : ', (⍕INSCASH[POS], ' , STILL OWING : ',⍕INSOWE[POS]
[9]  'INSTALMENT'
[10] INSCASH[POS]←INSCASH[POS]+V←⎕
[11] INSOWE[POS]←INSOWE[POS]-V
[12] →LO
[13] LEAVE:SAVEWS
     ∇
```

In [4] we look for every appearance of the employee in the vector *INSREG*; we obtain a binary vector called *CACHE*. It can only contain zeros if there has been a registration error, or if the registration is correct, but this employee has not requested an outing. Hence it remains to be seen, in [5 and 6], if the outing number introduced features in the list of outings requested by this employee. If this is not the case, the jump returns to the question *LO*.

If everything is correct, the term *OUT=INSOUT* AND the term *CACHE* (i.e. *INSREG=REP*[1]) enables the position *POS* of this registration to be found in the vectors *INS...*

It is quite easy to print the total already paid, and the amount still owing [8]. The instalment introduced in line [10] must be added to *INSCASH* and subtracted from *INSOWE* before restarting with the initial question by means of the jump [12].

READ BACK

We can explore the outings one after the other by a loop and calculate the total amounts paid, total amounts owing, etc ... This would require the same calculations to be repeated several times.

Furthermore, we are in good company and can skip a loop!

We start by arranging the vector *INSOUT* in increasing order of outing numbers. The result is called *IO*. In [2] we compare each term of *IO* with the following, which enables series of identical values to be detected. In [3], we deduce the position of the last value of each series.
For example:

if *IO* is the vector	:	3	3	5	5	5	5	6	8	8	8
IO ≠ 1↓*IO*,0 will equal	:	0	1	0	0	0	1	1	0	0	1
and *POS* will be equal to	:		2				6	7			10
if *INSCASH*[*INO*] equals	:	50	40	90	30	20	60	60	40	20	50
(+*IC*)[*POS*] will give	:		90				290	350			460

By shifting and subtracting we obtain : 90 200 60 110 i.e. the totals perceived by each outing.
These manipulations are represented by instructions [2 to 7].
The same operation is undertaken in [9] for the numbers of participants.

```
      ∇ TRIPS ;IO;IC;INO;POS;NB;BUD;PP;TEXT;BIN
[1]     IO←INSOUT[INO←⍋INSOUT]
[2]     BIN←IO≠1↓IO,0
[3]     POS← BIN/⍳⍴BIN
[4]     IC←+\INSCASH[INO]
[5]     IO←IO[POS]
[6]     IC←IC[POS]
[7]     IC←IC-0,¯1↓IC
[8]     NB←(+\INSB[INO])[POS]
[9]     NB←NB-0,¯1↓NB
[10]    BUD←NB×PP←OUTPRICES[OUTCODES⍳IO;]
[11]    TEXT←(((⍴IO),2)⍴' '),OUTNAMES[OUTCODES⍳IO;]
[12]    ''
[13]    'CODE          OUTING                    PRICE  PART  BUDGET   RECEIVED'
[14]    ''
[15]    (3 VERT IO),TEXT,5 0 5 0 8 0 8 0⍕PP ,NB ,BUD,[1.5]IC
      ∇
```

With *OUTCODES*⍳*IO* giving the indices of the outings, we can calculate the budget of each outing in [10 and 11] and extract its name.

It remains only to print the extracted values [15].
The vector of the outing numbers will be printed vertically by means of the *VERT* function written at the beginning of this work. The four vectors *PP*, *NB*, *BUD* and *IC* are joined into a matrix which is presented by means of a single FORMAT.
Note that a lamination is used to join *BUD* and *IC*; afterwards simple catenations suffice to join *NB* then *PP*.

The state of the individual instalments is fairly easy to obtain, since all the information is contained in the *INS .....,* vectors.

```
      ∇ STATE ;WHO;INO;INFOS;BIN;TEXT
[1]     WHO←INSREG[INO←⍋INSREG]
[2]     INFOS←(INSOUT,INSNB,0,INSCASH,[1.5]INSOWE)[IND;]
[3]     INFOS[;3]←+/INFOS[; 4.5]
[4]     BIN←WHO≠0,¯1↓WHO
[5]     TEXT←(3 VERT WHO),' ',PERNAMES[PERREGS⍳WHO;]
[6]     ''
[7]     'REG     NAME              OUT      INS   TOTAL   RECEIVED    OWES'
[8]     ''
[9]     (BIN\[1]BIN/[1]TEXT), 8 0 ⍕ INFOS
      ∇
```

Succesive grading arranges the registrations of the participants, so as to regroup all the requests of one and the same employee [1].

By the same method as above, we join all the *INS....* vectors into a matrix, which we classify in the same order as the registrations [2]. A column of zeros has been included; here we insert, in line [3] the total amounts already paid and still owing.

In *TEXT* we prepare the registrations and names of the employees (which are thus repeated as many times as there are outings per employee).

In order to avoid such repetitions, *BIN* receives the positions of the first appearances of a name. The technique used is the same as for *IV* on the preceding page. This binary vector enables a *TEXT* COMPRESSION to be undertaken so that each name is retained only once, and immediately afterwards an EXPANSION to provide blank lines [9].

This technique, which consists of looking for blanks, then using the same binary vector to carry out a compression and an expansion, is quite conventional. It can be used for the topic *"Good print-outs"*.

In order to print the participants, a loop is strongly advised. Each outing code is extracted [2] and placed in the variable *V*. *PART* receives the registration entered for this outing by means of a compression [3].

If there are no participants, the test on ρ*PART* passes to the next outing [4].

```
     ∇ PARTICIPANTS ;V;PART;POS;WHO;OUT;I;CACHE;DIM
[1]  I←1
[2]  L0: V←OUTCODES[I]
[3]  PART←(CACHE←INSOUT=V)/INSREG
[4]  →(0=ρPART)/L1
[5]  POS←PERREGS⍳PART
[6]  WHO←PERNAMES[POS;] , 2 VERT CACHE/INXNB
[7]  WHO←((3×⌈(ρPART)÷3),16)↑WHO
[8]  DIM←(1↑ρWHO)÷3
[9]  WHO←(DIM,48)ρWHO
[10] OUT←(DIM,30)↑ 1 30 ρOUTNAMES[I;]
[11] ''
[12] OUT,WHO
[13] L1: →((ρOUTCODES)≥I←I+1)/L0
     ∇
```

We calculate the indices of the participants' registrations by means of the INDEX-OF function [5], which enables the coresponding names to be extracted in [6]. We immediately join the numbers of participants to them.

In order to place three participants per line, the number of participants per employee must be a multiple of 3. Instruction [7] consists precisely of supplementing this list by blanks if necessary, by means of the TAKE function. We can thus re-shape the matrix obtained into a matrix which has one third the number of lines but three times more columns [8 and 9].

To print the outing name and the list of participants side by side we must see that these arrays have the same number of lines, whence instruction [10].

A loop explores all the outings one by one [13].

CONCLUDING AN OUTING

```
     ∇ ERASEOUT ;OUT;CACHE
[1]  L0: 'WHICH OUTING ?'
[2]  →(~(OUT←⎕)∊OUTCODES)/L0
[3]  CACHE←INSOUT=OUT
[4]  →(0∧.= CACHE/INSOWE)/OK
[5]  →0 AFTER 'UNSETTLED DONATIONS'
[6]  OK: PURGEOUT
[7]  SAVEWS
[8]  'IT''S DONE'
     ∇
```

A first test [2] provides for the incorrect codes to be detected, and return to the question [1].

A binary cache calculated in [3] enables the amounts still owing by the participants in this outing to be extracted [4]. It will be erased only if all these amounts equal zero.
We could write:

```
∧/ 0=CACHE/INSOWE
```

An inner product also undertakes this:

```
0 ∧.= CACHE/INSOWE
```

According to the result of this test, we either leave the function after having printed a message [5], or we jump to [6] where the function *PURGEOUT* takes control of updating the various variables.
A safeguard immediately follows:

```
     ∇ PURGEOUT
[1]  CACHE←~CACHE
[2]  INSREG ← CACHE/INSREG
[3]  INSOUT ← CACHE/INSOUT
[4]  INSNB  ← CACHE/INSNB
[5]  INSCASH← CACHE/INSCASH
[6]  INSOWE ← CACHE/INSOWE
[7]  CACHE OUTCODES≠OUT
[8]  OUTCODES←CACHE/OUTCODES
[9]  OUTNAMES←CACHE/[1]OUTNAMES
[10] OUTPRICES←CACHE/OUTPRICES
     ∇
```

A series of compressions updates the variables by means of two successive binary caches.

Still many more functions would be needed to complete this skeleton, to update the data concerning personnel, to modify the number of people registered by an employee, etc ...

CAN YOU UPDATE

First of all we must understand that the data we have do not give an exact picture of the life of the company, since the people engaged during the year and who left before the end of the year do not appear in the given vectors.

Having made this comment, the work is divided into two parts: calculating the updating matrix then the array of man-power transfer.

UPDATING MATRIX

The *UPDATE* function performs this task.

We first look for the position of *REG*1 registration in *REG*2. For employees still present we obtain a normal value. For employees who have left, we obtain 1+ρ*REG*2 or again 1+ρ*CAT*2 (line [1]).

We obtain the new category of employees by *CAT*2[*POS*]. However, owing to the employees who left, we have written (*CAT*2,0)[*POS*] so as to obtain the value 0 for the people who have left [2].

BIN must enable us to detect engagements. We give it the dimensions of the size of *REG*2 plus one [3]. The *POS* indices indicates the people who were present last year. These are not engagements, and we place a zero in *BIN*[*POS*] (line [4]). It is quite easy from this to deduce the categories of people engaged [5].

```
      ∇ R←UPDATE ;POS;NEWCAT;BIN;ENGAGEMENTS;INICAT;FINCAT
[1]   POS←REG2⍳REG1
[2]   NEWCAT←(CAT2,0)[POS]
[3]   BIN←(1+⍴REG2)⍴1
[4]   BIN[POS]←0
[5]   ENGAGEMENTS←BIN/CAT2,0
[6]   INICAT←CAT1,(⍴ENGAGEMENTS)⍴0
[7]   FINCAT←NEWCAT,ENGAGEMENTS
[8]   R←INICAT  CROSSCOUNT  FINCAT
      ∇
```

In order to calculate the transfers, each person must have an initial category (*INICAT*) and a final category (*FINCAT*). For people engaged, we will place their initial category at zero [6]; for people who have left, *NEWCAT* already contains zeros by way of the final category, but the people engaged must be integrated [7].

Two vectors are thus obtained, *INICAT* and *FINCAT*, of the same length, which we can submit to an auxiliary function *CROSSCOUNT*, so as to calculate the transfer matrix.

THE *CROSSCOUNT* FUNCTION

There are several ways of writing the *CROSSCOUNT* function. Here is one of them, directly inspired by the crossed numbering functions which are the subject of the topic *"Crossed numbering"*. Consult the corresponding solutions.

```
      ∇ R←A CROSSCOUNT B ;B BINV;BINH
[1]   BINV ← (0,CATEGORIES)∘.=A
[2]   BINH ← B∘.=(CATEGORIES,0)
[3]   R ← BINV +.∧ BINH
      ∇
```

CATEGORIES is a global variable which contains the different values which the category can take.

Here is a more traditional method:

```
      ∇ R←A CROSSCOUNT B ;I
[1]   R ← (2ρ1+ρCATEGORIES)ρ0
[2]   A ← (0,CATEGORIES)⍳A
[3]   B ← (CATEGORIES,0)⍳B
[4]   I ← 1
[5] L0: R[A[I] ; B[I]] ← 1+R[A[I] ; B[I]]
[6]   →((ρA)≥I←I+1)/L0
      ∇
```

Here is a third method which, before consituting the final matrix, consists of counting the people who have had the same type of development during the year:

```
      ∇ R←A CROSSCOUNT B;DEV;BIN;TOT;POS;VEC;DIM
[1]   DEV←1+(DIM←1+ρCATEGORIES)⊥A,[0.5]B
[2]   DEV←DEV[⍋DEV]
[3]   BIN←DEV≠1↓DEV,0
[4]   TOT←BIN/⍳ρBIN
[5]   TOT←TOT-0,¯1↓TOT
[6]   POS←BIN/DEV
[7]   VEC←(DIM×DIM)ρ0
[8]   VEC[POS]←TOT
[9]   R←1⌽(DIM,DIM)ρVEC
      ∇
```

This method does not require as much space as the first (no outer product), and is much quicker than the second (no loop). The development of a person is shown by a whole number [1-2]. When looking for value changes [3], we find the number of people who have had a given development [4-5]. It suffices to place these values in a vector which is the image of the desired result ravelled. This is why the calculation of *DEV* is undertaken in such a way as to directly give the index of development of employees in this vector *VEC*.

These three, very different solutions, clearly demonstrate the richness of APL. The first is elegance itself, but the last is less greedy!

SECOND STEP

With the updating matrix already known, the *CALTRANS* function serves to calculate the array of man-power transfers. It has 8 lines, and as many columns as categories (the total will be added in line [9]). The *TRANSFERS* function serves only to link the two sub-functions, and to present the results correctly:

```
     ∇ TRANSFERS ;STATE
[1]  'UPDATING MATRIX'
[2]  ''
[3]  ⎕←STATE←UPDATE
[4]  ''
[5]  'ARRAY OF MAN-POWER TRANSFERS'
[6]  ''
[7]  (16↑'CATEGORIES'), 4 0 ⍕CATEGORIES
[8]  ''
[9]  I 0 1 1 1 0 1 1 1 0 1 \[1] TEXT, 4 0 ⍕CALTRANS
     ∇
```

TEXT is a characters matrix containing the lables of the final array.

THE *CALTRANS* FUNCTION

```
     ∇ R←CALTRANS ;CACHE
[1]  R←(8,⍴CATEGORIES)⍴+/ 1 0 ↓STATE
[2]  R[3;]←¯1↓STATE[1;]
[3]  R[6;]←1↓,STATE[;1+⍴CATEGORIES]
[4]  CACHE←0,[1] (CATEGORIES∘.<CATEGORIES),0
[5]  R[2;]←¯1↓ +/[1] STATE×CACHE
[6]  R[5;]← 1↓ +/[2] STATE×CACHE
[7]  R[4 7;]←R[2 5;]+R[3 6;]
[8]  R[8;]← +/[1] STATE[;⍳⍴CATEGORIES]
[9]  R←R,+/R
     ∇
```

The first line must contain the initial man-power. This is the total of the values featuring in the lines corresponding to each category [1]. The final man-power is the total of the columns (instruction [8]).

The engagements are indicated directly by the first line of the *STATE* array, last element excluded. In the same way, permanent departures are obtained by the last column of the *STATE* array, first element excluded. This clarifies instructions [2 and 3].

A binary matrix called *CACHE* serves to extract the grater right hand part of the array, which contains the category changes [4]. Entries by promotion are obtained by adding this extract vertically [5], the outgoings by promotion are obtained by the horizontal sum in instruction [6].

Instruction [7] calculates the total arrivals and departures.

SPECIAL PRINTING

GENERAL PRINCIPLES

The final read back will be formed by successive catenations of the columns representing each datum. These columns will be separated from one another by the three characters ' | '. This is the role of the matrix *SEP*.
The header and the horizontal dotted line on the bottom are added last.

TABLES

In order to print the data conveniently, we must know:

1 - the manner of selecting the information concerning a list of individuals, referenced by their numbers *NOS*:

- indexing by [*NOS*] for the vectors such as (*AGE* or *SEX*),
- indexing by [*NOS*] for the matrices (such as name).

2 - the printing format of these data:

- a characters matrix does not require treatment,
- a numeric matrix must be transformed into a vertical characters matrix by means of the *VERT* function,
- a character vector must be written vertically, which is performed automatically by catenating to *SEP*.

All this information could be catenated automatically but repetition of these calculations at each printing would be costly and would burden the functions. We have preferred to use a table of formats, *TAFOR*, constituted manually or calculated once for all of them by a preliminary function.

```
      NAME      [NOS;]
    2 VERT AGE  [NOS]
      SEX       [NOS]
    2 VERT DEPT [NOS]
      MARSTA    [NOS]
    3 VERT REG  [NOS]
```

TAFOR is a matrix of 6 lines and 20 columns. Columns 9 to 14 are reserved for the names of the variables.

It would be very difficult to constitute the headers automatically.

We have constituted a table of headers concerning each item. This table is called *TAHED*. It has 6 lines and 20 columns. Each header is bound there by the symbol ⌊. Of course, there must be parity between the resulting width of the format of a zone and the width of its header. Here are the contents of *TAHED*:

```
      N A M E   |
  AGE |
  SEX|
  DEPT|
  M.S.|
   REG |
```

THE *PRINT-OUT* FUNCTION

```
      ∇ R←PRINT-OUT NOS ;SEP;Z;I
  [1]  ←'NOBODY CORRESPONDS TO THIS DEFINITION'
  [2]  →(0=ρNOS←,NOS)/0
  [3]  SEP←((ρNOS),3)ρ ' | '
  [4]  R ← 0 1 ↓ SEP
  [5]  I ← 1
  [6] L0: Z ← ZONES[I]
  [7]  R ← R,(⍎TAFOR[Z;]),SEP
  [8]  →((ρZONES)≥ ← +1)/L0
  [9]  R ← HEADER,[1](0 ¯1 ↓ R),[1] HEADER[3;]
      ∇
```

Instructions [1 to 3] are clear. In [4] we initialise the result by a column of vertical dashes bordered by a blank column.

We then explore the zones one after another [6] and we execute the corresponding line of *TAFOR*. The result of this is a matrix of characters which represents the corresponding information. We catenate it to *R*, and follow it by the matrix *SEP* to terminate the column (instruction [7]).

When all the zones to be printed have been explored [8], it remains only to remove the last blank column left by *SEP* from *R*, to cap the result by *HEADER*, and to conclude at the bottom by *HEADER* [3;].

Wasn't that easy?

ZONES and *HEADER* are elaborated by the functions *DEFZONES* and *DEFHEADER* respectively, called by the function *CHARACTER*. The latter is quite clear.

```
      ∇ CHARACETR
  [1]  DEFZONES
  [2]  DEFHEADER
  [3]  ''
  [4]  '----> IT'S DONE.'
      ∇
```

THE *DEFZONES* FUNCTION

```
     ∇ DEFZONES ;ZO;NB; VARS
[1] L0: ⍞←25↑'DATA TO BE PRINTED ......'
[2]  ZO ← 25↓⍞
[3]  VARS ← 6 0⍴''
[4] L1: ZO ← (∨\ZO≠' ')/ZO
[5]  NB ← ¯1+ZO⍳' '
[6]  VARS ← VARS,6↑NB↑ZO
[7]  →(0<⍴ZO←NB↓ZO)/L1
[8]  ZONES ← (⍳1 ⍴TAFOR)+.×TAFOR[;8+⍳6]∧.=VARS
[9]  →(~0∊ZONES)/0
[10] →L0 AFTER 'UNKNOWN DATA  : ',,⍉(ZONES=0)/VARS
     ∇
```

ZO is the vector typed by the user [1-2]. A loop enables the header blanks to be eliminated [4], the length *NB* of the first word to be found [5], and enables it to be catenated to a matrix *VARS* initialised in [3].
The operation is facilitated by the fact that the names of variables have 6 characters maximum.
We obtain, for example, the matrix *VARS* as follows:

```
REG
NAME
DEP      (error!)
AGE
```

TAFOR[;8+⍳6] contains the names of the authorised variables. The classic inner product ∧.= gives one binary column per word typed. Multiplying matricially by ⍳6 we obtain:

```
6 1 0 2
```

This result shows that the word *DEP* is unknown, whereas *REG*, *NAME* and *AGE* are zones 6, 1 and 2 respectively.
Instructions [9 and 10] serve to attract attention to the unknown words.
The function leaves the global variable *ZONES* in the workspace.

THE *DEFHEADER* FUNCTION

```
     ∇ DEFHEADER ;BIN;POS
[1]  HEADER ← TAHED[ZONES;]
[2]  BIN ← , ⌽∨\⌽HEADER='|'
[3]  HEADER ← '|', BIN/,HEADER
[4]  POS ← (HEADER='|')/⍳⍴HEADER
[5]  HEADER ← '¯',[1] HEADER,[0.5] '_'
[6]  HEADER[;POS] ← '|'
     ∇
```

We start by extracting the lines of *TAHED* corresponding to the zeros to be printed [1].

In order to eliminate surplus blanks, we turn the table and apply the method shown in paragraph 5-2 of the course, page 113. Ravelling the result gives a binary vector which serves to compress the vector, *HEADER* (instruction [3]).

It suffices therefore to transform *HEADER* into a matrix by lamination with an underline of dashes, then to add an overline of dashes by catenation [5].

Instructions [4] and [6] enable the vertical dotted lines ⁞ to be extended from top to bottom.

The selection and classification functions were shown in detail in the course, pages 361 to 363.

It is seen that the functions required are relatively short and simple. They are very general and can come to the rescue in many everyday situations. They are, moreover, pleasant to use even by people knowing nothing about data processing.

AHEAD OF THE CLIENT

DATA STRUCTURE

The structure adopted for the data is justified by considerations of ease of use and security.

DATA DIMENSIONING

Most users dimension their data to the minimum required. For example, in order to contain the 13 clients of our diagram, they would use vectors of 13 elements, which they would enlarge afterwards by means of successive catenations. This is a saving which appears greater than it actually is and in fact it leads to a lot of waste and difficulty.

- Apparent saving only, since some day the size of these data will have to be increased, to meet the arrival of new clients. At final reckoning, we will not have saved anything. Why not reserve sufficiant space, in one go, to accommodate expansions foreseeable in the short term?

- Waste and difficulties, because if a client leaves, his place will have to be recuperated at the price of logical compressions, of compacting files, all VERY costly operations.

On the other hand, the structure adopted enables a new client to be inserted by simply indexing (a simple and cheap operation), and a client to be erased simply by placing a zero in a bnary vector. Try it. It is much more economical.

PROCESSING SECURITY

We will imagine that a new client arrives. The various data concerning him must be updated. If an incident interrupts the course of this updating, certain data will contain only 13 of them. In data processing, such a situation is always very uncomfortable.

With the structure adopted here, we will update *BINCLI* at last. If one position of *BINCLI* equals 1, we are assured that all the information concerning this client has been introduced. If it equals zero, the corresponding space will be deemed empty, even if updating has been started. A client is thus introduced totally, or not at all. He can never be half way.

It is certainly the most reliable structure in the event of an incident.

FIRST STEP

THE *INTROCLI* FUNCTION

CLI is the highest code at the moment [1]; we will increase it at each introduction of a new client [3]. *NB*, length of the vector *CLICO* will be used by the function *GETINA*.

The slave function *GETADD* reinstates in *AD* an address of 120 characters, or an empty vector [4]. If this address is empty [5], we disconnect in [13], at the end of the processing. We would normally find a safeguarding process on file there, which was not requested here.

The function *STORE* places this address in the first empty line of *ADDRESSES*, and reinstates the index of this line. This index represents the position of the client in the data (*POCLI*), and will be taken temporarily as the invoicing address (*POINV*), for the case where the two addresses would be identical.
The client code is placed in this position [7].

```
        ∇ INTROCLI ;CLI;AD;POCLI;POINV;BG;NB;FIGURES
[1]     CLI←⌈/CLICO
[2]     NB←⍴CLICO
[3]     NEXT: □TC[3],'----> INTRODUCTION OF CLIENT  ',⍕CLI←CLI+1
[4]     AD←GETADD 'DELIVERY '
[5]     →(0=⍴AD)/EXIT
[6]     POCLI←POINV← 0 STORE AD
[7]     CLICO[POCLI]←CLI
[8]     →GETGROUP/TAIL
[9]     GETINA
[10]    TAIL: INVAD[POCLI]←POINV
[11]    BINCLI[POCLI,POINV]← 1
[12]    →NEXT
[13]    EXIT: '----> END'
        ∇
```

In line [8], the result of the function *GETGROUP* according to the case enables a jump to [10], or the invoicing address to be requested, by means of the *GETINA* function; instruction [9].

GETGROUP or *GETINA* having possibly modified the pointer *POINV*, we can place it in position *POCLI* of the *INVAD* vector; instruction [10].

The client is completely introduced, we can update *BINCLI* and return to line [3].

Now we will study the slave functions.

THE *GETGROUP* FUNCTION

This function serves to determine if the client belongs to a buying group. In order to be able to reply by a carriage-return, we must obligatorily use a ⍞ and not a ⎕ (see course page 106). Here, we have used even the "bare-output" technique so that the answer is on the same line as the question (see course page 107). This is the meaning of instructions [1 to 3].

If none of the characters typed is numeric, we leave the function, and *R* takes the value zero [4]. If not, these numeric characters are transformed into a scalar *BG* by the function ⍎ , whereas *R* keeps the value 1; instruction [5].

```
        ∇ R←GETGROUP ;BG;NUM
[1]    BUY: ''
[2]     ⍞←'BUYING GROUP ...'
[3]     NUM←(BG←⍞)∊'0123456789'
[4]     →(~R←∨/NUM)/0
[5]     BG←⍎ NUM/BG
[6]     →(BG≠CLI)/SEARCH
[7]     GROUPS[POCLI]←1
[8]     → 0 AFTER 'GROUP RECORDED'
[9]     SEARCH: →(~BG∊GROUPS/CLICO)/ERROR
[10]    POINV←CLICO⍳BG
[11]    →0
[12]    ERROR: → BUY AFTER 'THIS IS NOT A B.G. CODE'
        ∇
```

If *BG* is identical with the client code, it is because one wished to create a buying group. We note it in the vector *GROUPS*, then we exit after confirmation [7 and 8].

If not, we ascertain whether the code typed is indeed that of a group already recorded. According to case, either we print an error message [12], or we note in *POINV* the index of this group in *CLICO*; see [10].

THE *GETINA* FUNCTION

```
        ∇ GETADD ;AD
[1]     AD←GETADD 'INVOICING'
[2]     →(0=⍴AD)/0
[3]     POINV←NB STORE AD
        ∇
```

If no address is introduced, we leave the function without processing [2].

If not, we store this address in the first free space after the first *NB*'s which are reserved for delivery addresses. We note in *POINV* at which index this address was placed.

THE *GETADD* FUNCTION

Here again we have used the bare-output technique. The processing of the first line of address has been separated from the processing of the following three, so as to detect the possibility of an empty reply.

```
      ∇ R←GETADD TEXT ;I
[1]   ⎕TC[3], 'ADDRESS OF',TEXT
[2]   ⍞←6⍴' '
[3]   →(0=⍴R←6↓⍞)/0
[4]   R←30↑R
[5]   I←1
[6]  L0: ⍞←6⍴' '
[7]   R←R,30↑6↓⍞
[8]   →(3≥I←I+1)/L0
      ∇
```

Remember that ⎕*TC*[3] represents the line feed on IBM systems.

THE *STORE* FUNCTION

```
      ∇ R←ORIGIN STORE AD
[1]   R←ORIGIN + (ORIGIN↓BINCLI)⍳0
[2]   ADDRESSES[R;]←AD
      ∇
```

We first look for the position of the first zero of *BINCLI* after the given origin: 0 for delivery addresses, or *NB* for invoicing addresses. Having found this position, we place the address *AD* in the corresponding line of *ADDRESSES*.

Each one of these functions plays a precise role. We thus end up with functions which are short, easy to modify and simple to read back. Some, such as *GETADD*, can be useful in other circumstances.

Such an approach to the problems is infinitely preferable to the single function, instructed to de everything, which becomes illegible.

SECOND STEP

The header of the print-outs is constituted by a global variable *HEDCLI*.
After printing this header [6 and 7], lines [8 to 11] serve to print each client in succession. The slave function *PRINTCLI* carries out the presentation.

```
        ∇ PRICLI ;SEP;OFT;DITTO;POCLI;NB;I
[1]     SEP← 4 3 ρ'|'
[2]     OFT← ¯3 30 ↑ 1 21 ρ'          -- GROUP --'
[3]     DITTO← 4 30 ↑ 1 21 ρ'         SAME ADDRESS'
[4]     POCLI←((ρCLICO)↑BINCLI)/ιρCLICO
[5]     NB←ρPOCLI←POCLI[⍋CLICO[POCLI]]
[6]     ''
[7]     HEDCLI
[8]     I←1
[9]     B: ''
[10]    PRINTCLI POCLI[I]
[11]    →(NB≥I←I+1)/B
        ∇
```

The existing clients are known by the 1's of the first part of *BINCLI*. We therefore seek their positions [4], then arrange them in terms of their codes [5].
The numbers *NB* of clients serves, in line [11], to count the repetitions in the loop.

Certain matrices of characters must be printed very often. It would be clumsy to create them in *PRINTCLI*, because they would be regenerated at each passage in this function. Hence it is better to create them in *PRICLI*. They then become global for *PRINTCLI*.

These variables, *SEP*, *OFT* and *DITTO* are created in lines [1 to 3].

THE *PRINTCLI* FUNCTION

```
        ∇ PRINTCLI POS ;LINE;INA;INV
[1]     LINE←(' G'[1+GROUPS[POS]], 6 0⍕CLICO[POS]),[1] 3 7ρ' '
[2]     LINE LINE,SEP,(4 30 ρ ADDRESSES[POS;]),SEP
[3]     INA←DITTO
[4]     INV←INVAD[POS]
[5]     →(INA=POS)/EXIT
[6]     INA← 4 30 ρ ADDRESSES[INA;]
[7]     →((GROUPS,0)[INV⌊ 1+ρGROUPS]=0)/EXIT
[8]     INA[2 3 4 ;]←OFT
[9]     INA← 2 ⊖  INA
[10]    EXIT: LINE,INA,SEP
        ∇
```

For each client, we join the following into a big matrix.

- its client code, possibly preceded by the letter *G* if it concerns a buying group [1].
- its delivery address, separated from that which precedes it by the separator *SEP*; instruction [2].
- its invoicing address. Since it is often identical with the delivery address, we have placed a matrix *DITTO*, which contains the message "*SAME ADDRESS*" in the variable *INA*. This is instruction [3].
  In *INV* we note the index of the invoicing address [4]. If it is effectively identical to the delivery address, we can disconnect in [10] for final printing.

  If, on the other hand, the two addresses are different, we start by extracting the invoicing address and structuring it into a matrix [6].
  However, if the client belongs to a buying group, only the company name of the latter should be printed. To arrive at this, we quash lines 2, 3 and 4 of the address by the matrix *OFT*, which contains the message"-- *GROUP* --" (instruction [8]). Hence a rotation gives the desired presentation [9].

In all cases, we end by putting the variable *LINE*, followed by the invoicing address, followed by the separator *SEP*, which terminates the presentation [10].

We will now return to instruction [7] which serves to determine whether the client belongs to a buying group. The expression *GROUPS* [*INV*] would be highly suitable, except for clients having their invoicing address outside the limits of *GROUPS*, whence the formulation adopted here.

THIRD STEP

An auxiliary function, *GETCLI*, enables a client to be serched for, displays the name and requests validation. It gives the index of this client as result [3].
If this client is not a group [4], it will be erased immediately by placing a zero in *BINCLI* at the positions of the delivery and invoicing addresses [12].

On the other hand, in the case of a buying group, we must look for the list of its members, called *MB* here. These members are the clients:

- who have not yet left, i.e.: *NB↑BINCLI*
- whose invoicing address registers on the erased client, i.e.: *INVAD=POS*

- finally, who are not the group itself: $(\iota NB) \neq POS$

The combination of these conditions gives, in line [5], the binary vector *BIN* which enables the list of members to be known.

We display these codes [6-7], then we calculate the positions of these clients in the file. This is *POMB*. Hence we must replace, in *INVAD*, the buying groups address by that of the clients, i.e. *POMB* precisely. This is what is done in line [9].

We must also place a zero in *GROUPS*[*POS*], because this position will be re-used by a new client one day [10].

```
        ∇ ERASECLI ;CLI;POS;POMB;BIN;MB;NB;END
[1]       END←*1
[2]       NB←⍴CLICO
[3]       NEXT: →(0=POS←GETCLI)/0
[4]       →(GROUPS[POS]=0)/ERASE
[5]       MB←(BIN←(NB↑BINCLI)∧(INVAD=POS)∧(POS≠⍳NB))/CLICO
[6]       ''
[7]       'THIS BUYING GROUP CONCERNS CLIENT  :  ',⍕MB
[8]       POMB←BIN/⍳NB
[9]       INVAD[POMB]←POMB
[10]      GROUPS[POS]←0
[11]      ERASE: POINV ←(INVAD[POS]>NB)/INVAD[POS]
[12]      BINCLI[POS,POINV]←0
[13]      '----> CLIENT ERASED'
[14]      →NEXT
        ∇
```

<u>N.B. :</u>

If we erase a member client of a buying group, we must take care not to place a zero against the invoicing address in *BINCLI*, because this results inerasing the address of the buying group.

Also *POINV* is calculated only if it concerns an address beyond the first *NB* [11].

<u>THE *GETCLI* FUNCITON</u>

The function first extracts the list of clients still present [1], then requests the introduction of the code of a client [2].

If we answer *END*, we leave the function, which gives the result 0.

If not, the function checks that the client exists in [4], absence of which causes an error message to be emitted [5].

```
      ∇ R←GETCLI ;BIN;CLI;CODES
[1]     L0: CODES←(NB↑BINCLI)/CLICO
[2]     ⎕TC[3],'CLIENT CODES'
[3]     →(END=CLI←⎕ )/R←0
[4]     →(NB≥R←CLICO⍳CLI)/EXISTS
[5]     →L0 AFTER 'THIS IS NOT A CLIENT'
[6]     EXISTS: ⍞←ADDRESSES[R;⍳30],' / VALIDATE.......'
[7]     →('N'∈⍞ )/L0
      ∇
```

If the client exists, *R* takes its index in the data for a value.

In order to check the name, we display, in line [6], the first 30 characters of the address, by the bare-output method, already used. The simplest and most efficient means of using the answer consists of detecting the presence of the letter *N* in the answer.

- if the user replied *NO*, *NON*, *NIET* or *NEIN*, the presence of the letter *N* sufficies for it to be established that it is not in agreement on the identity of the client. Hence we return to the beginning of the function.
- if the user replies *YES*, *OUI*, or typed a simple carriage-return, the absence of *N* suffices to validate the client code.

If we wish for greater security, we have to check that the user has answered precisely *YES*.
We could thus use the expression:

```
'YES' ∧.= 3↑(REP≠' ')/REP←⍞
```

Generally, however, the method advised above is quite adequate, and far less restrictive for the user.

BLOCK-HEADS

NUMBER will contain the number of blocks to be plotted [1].

Since the left-hand argument can have either one or two values, we obtain, by the expedient of instruction [2], a vector of two values, of which the second is either 5 or the value used as argument.

We multiply the values given by the vertical scale, and take the rounded figure. Moreover, as it is easier to count numbers of lines starting at the top of a matrix, we will plot the diagram upside down, and return it at the end of the processing. That is why we reserve the values supplied by ⊖ [3].

The positions of the horizontal lines are thus given by +\[1]*REP* increased by 1, since we must plot a horizontal dotted line in order to conclude the graph at the bottom [4].

For the same reason, the heights of the blocks are given by 1++/[1]*REP* and the total height of the diagram is given by the height of the biggest block: *MAX* ← ⌈/*HEIGHT* (instruction [5]).

```
      ∇ R←SCA BLODIAG REP ;NUMBER;WIDTH;HEIGHT;LINES;MAX;LI;COL;BNO
[1]     NUMBER ← ¯1↑⍴REP
[2]     WIDTH ← (SCA←2↑SCA,5)[2]
[3]     REP ← ⊖ ⌊0.5+SCA[1]×REP
[4]     LINES ← +\[1] 1,[1]REP
[5]     MAX ← ⌈/HEIGHT ← 1+ +/[1]REP
[6]     HEIGHT ← (0,HEIGHT)⌈(HEIGHT,0)
[7]     R ← (MAX,1+WIDTH×NUMBER)⍴ ' '
[8]     R[⍳HEIGHT[1];1]←'|'
[9]     COL ← 1+⍳WIDTH
[10]    BNO ← 1
[11]    L0: LI ← LINES[;BNO]
[12]    R[LI;COL]←'-'
[13]    R[⍳HEIGHT[BNO+1];1+¯1↑COL]←'|'
[14]    COL ← COL+WIDTH
[15]    →(NUMBER ≥ BNO←BNO+1)/L0
[16]    R ← ⊖R
      ∇
```

If there are 6 blocks, there are 7 vertical separation dotted lines. Their height is equal to the height of the biggest of the two intermediate blocks [6].

We can thus define an array of characters, *R*, the height of *MAX*, and of width 1+*NUMBER*×*WIDTH* (instruction [7]). Its first column is a vertical dotted line of height *HEIGHT*[1] (instruction [8]).

We now place the horizontal and vertical dotted lines block by block by means of a loop. *BNO* is the number of the block plotted. We make it vary from 1 to *NUMBER* (instructions [10] and [15]).

COL represents the positions of the horizontal dotted lines in the columns of *R*. There are *WIDTH*-1 of them. *LI* gives the height at which these dotted lines must be placed [11].

We thus write dashes in the lines *LI* and the columns *COL* in instruction [12].
The following column will contain a height dotted line *HEIGHT*[*BNO*+1]

It therefore suffices to increase the indices of columns by the width of one block [14], and to recommence upwards to the last block.
In the last instruction, we re-erect the diagram, which was upside down.

The function obtained is considerably more complex than the function *MULTIBLO*, already written.

FOR THE REALLY KEEN

It is possible to do better still. We will imagine that the global variable *HEDWO* is a matrix of characters containing words relating to the lines of the right-hand argument:

```
FRANCE
ITALY
SWITZERLAND
```

We can consider inserting these references into the boxes thus:

```
|--------------|
| FRANCE       |               |--------------|
|--------------|--------------|              |
|              |              | FRANCE       |
|              |              |              |--------------|
|              | FRANCE       |              |              |
|              |              |--------------|              |
|              |--------------| ITALY        | FRANCE       |
| ITALY        |              |              |              |
|              |              |--------------|              |
|              | ITALY        |              |              |
|              |              |              |--------------|
|--------------|              |              | ITALY        |
|              |              |              |              |
|              |--------------|              |--------------|
|              |              | SWITZERLAND  |              |
| SWITZERLAND  | SWITZERLAND  |              | SWITZERLAND  |
|              |              |              |              |
|--------------|--------------|--------------|--------------|
```

Don't you feel an irresistible urge to solve this problem?

OIL TO THE FLAMES

DATA STRUCTURE

The vectors *CODE* and *PRICE* and the matrix *PRODUCT* have already been defined. For, reasons quoted in the topic "*Ahead of the client*", these data will have fixed dimensions. Some positions will be occupied by bits of information, whereas others will be still unused. A binary vector, *BINPRO*, will indicate the occupied places (1) or free places (0).

In order to represent the compositions of the derivatives, we will use a second set of data.
The vector *DERIVCO*, of variable length, contains the codes of the derivatives, whose price will have to be calculated at each modification.

An array of three dimensions, *COMPOS*, contains, in the order indicated by *DERIVCO*, and for each derivative:

- in its first plane *COMPOS*[1;;] the QUANTITIES of each product used in its composition.
- in its second plane *COMPOS*[2;;] the CODES of the products which it comprises.

This array is 10 columns wide, since a product cannot have more than ten components.

Here, the initial values are as follows:

DERIVCO	*COMPOS*										
210	83	17	0	0	0	0	0	0	0	0	QTES
320	85	15	0	0	0	0	0	0	0	0	
337	3	90	7	0	0	0	0	0	0	0	
	207	208	0	0	0	0	0	0	0	0	CODES
	209	318	0	0	0	0	0	0	0	0	
	318	332	333	0	0	0	0	0	0	0	

It is seen in this set of data that product 320 comprises 85% and 15% of products 209 and 318 respectively.

A similar solution would have consisted of nothing in the second plane, not the codes of the products but their INDICES in the vector *CODE*. This can obviate some calculations, but in the event of error, or during the set-up phase, a code is easier to interpret manually.

PROCESSING FUNCTIONS

PRINT-OUT OF THE COMPLETE CATALOGUE

When a product is erased, we place a zero in the vector *BINPRO* but its code continues to be present in the vector *CODE*. The place thus left empty will doubtless be re-used by a product whose code will be higher. Hence the codes are in any order. The position of the codes to be printed is obtained in [1] and the order in which they will have to be printed is obtained by classifying [2].

Simple indexing enables the information required to be extracted in a suitable order. The two numeric vectors are printed vertically by means of the auxiliary function *VERT*, written at the beginning of this book.

```
      ∇ PRINPROD ;POS;C;P
[1]     POS←BINPRO/⍳⍴BINPRO
[2]     POS←POS[⍋CODE[POS]]
[3]     ''
[4]     'CODE         PRODUCT              PRICE'
[5]     ''
[6]     C←4 VERT CODE[POS]
[7]     P←7 VERT PRICE[POS]
[8]     C, ' ',PRODUCT[POS;],P
      ∇
```

INTRODUCTION OF NEW PRODUCTS

The first step consists of placing the highest code number in *C*. We then increase this value 1 by 1 in order to assign a code to the products introduced.
A loop then displays the code of the product to come [5]. We have used the "bare-output" technique so that this number is on the same line as the description.
If this description *DESCRIBE* is empty, it is because the user has finished [6].

If not, we must look for the first empty place in the data. It is given by the first zero featuring in *BINPRO*; instruction [7].

We can now introduce the code and decription in the suitable positions [8 and 9]. Beware, however: it may be that no place in question has been occupied before by another product, which has since disappeared. Its price thus remains in place , and we must take care to cancel it in order to avoid confusion, whence instruction [10].

```
        ∇ INTROPROD ;C;DESCRIBE;POS
[1]     C←⌈/CODE
[2]     ''
[3]     'CODE        DESCRIPTION'
[4]     ''
[5]     L0: ⍞←6↑ 4 0 ⍕ C←C+1
[6]     →(0=⍴DESCRIBE←6↓⍞ )/0
[7]     POS←BINPRO⍳0
[8]     PRODUCT[POS;]←20↑DESCRIBE
[9]     CODE[POS]←C
[10]    PRICE[POS]←0
[11]    BINPRO[POS]←1
[12]    →L0
        ∇
```

When updating is finished, a 1 in *BINPRO* concludes the work (instruction [11]) and a loop continues [12].

If an interruption should occur between instructions [9] and [11], and the data is safeguarded in this state, the last code introduced would be present, although not corresponding to any actual product.
The following product would be assigned the code immediately higher.

INTRODUCING OR UPDATING COMPOSITIONS

An auxiliary function *VERIPROD* questions the user, and registers in *PRO* either the code of the product whose composition he wants to introduce, or zero if he has finished [1].

In [2 and 3] we introduce the composition of the product. Provision is made here so that we can withdraw by typing *END*, which is a luxury precaution. *END* is a global variable which equals *1.
In [4 and 5] we must check that we have introduced an EQUAL number of values. If this is indeed the case, we can structure them into a matrix of 2 lines: one line of quantities and one line of product codes [6].

In order to establish whether the components indicated indeed exist, we must see if they belong to *BINPRO/CODE*. In the event of an anomaly, the unknown codes will be displayed. This is achieved by instructions [7 and 8].

A test [9] serves to determine whether the derivative already features in the list of derivatives. If this is not the case, it must be added to it (instruction [10]), and a line of zeros added to the two planes of the *COMPOS* array (instruction [11]).

In [12], we calculate the index of the derivative in the vector *DERIVCO*.

It remains only to place the composition introduced in this line [13].

At the end of the work [15] an auxiliary function, *CALPRICE*, recalculates ALL the prices. This function will be studied in detail later.

```
      ∇ UPDCOMP ;CP;PRO;COMP;BIN;LI
[1]   L0: →(=0PRO←VERIPROD)/L5
[2]   L1: 'COMPOSITION [END]'
[3]   →(END∊COMP←,⎕ )/L0
[4]   →(0=2|⍴COMP)/L2
[5]   →L1 AFTER 'UNEQUAL NUMBER OF VALUES'
[6]   L2: COMP←⍉ ((0.5×⍴COMP),2)⍴COMP
[7]   →(∧/BIN←COMP[2;]∊BINPRO/CODE)L3
[8]   →L1 AFTER'PRODUCTS UNKNOWN : ',⍕(~BIN)/COMP[2;]
[9]   L3: →(PRO∊DERIVCO)/L4
[10]  DERIVCO←DERIVCO,PRO
[11]  COMPOS←COMPOS,[2] 0
[12]  L4: LI←DERIVCO⍳PRO
[13]  COMPOS[;LI;]← 2 10↑COMP
[14]  →L0
[15]  L5: CALPRICE
      ∇
```

In line [3] note the ravelling, ⎕ for the case where the user would introduce only a sinlge value.
Also, in line [4], note the use of the RESIDUE function (|) to determine the remainder of the division of ⍴*COMP* by 2.

THE *VERIPROD* FUNCTION

This function checks that a product code typed on the keyboard indeed exists. N.B: some codes may correspond to products which have since been erased, whence the compression by *BINPRO*.

```
      ∇ R←VERIPROD ;CP
[1]   L0: LF,'PRODUCT CODE [END]'
[2]   →(END=CP←''⍴⍞ )/R←0
[3]   →(CP∊BINPRO/CODE)/L1
[4]   →L0 AFTER 'PRODUCT UNKNOWN'
[5]   L1: ⍞←PRODUCT[CODE⍳R←CP ;],' ; VALIDATE.....'
[6]   →('N'∊⍞ )/L0
      ∇
```

Note that in order to provide for the introduction of a vector, '' ⍴⎕ gives a scalar.

In [5] the "bare-output" technique enables us to display the product name and to await the reply on the same line. If the user types *NO* or *NON* or *NIET*, etc ... the presence of *'N'* in his answer returns to *L0*. If not, we assume that he accepts the product as being correct [6].

PRINTING THE COMPOSITION OF DERIVATIVES

The printing order is increasing order of the vector *DERIVCO* (instruction [2]). We then take the derivatives one by one, starting with the first [3].

POS is the position of a derivative in *CODE*. We extract its composition (percentages and component products) in [5]. We then eliminate the zero values from this [6].

```
      ∇ PRINCOMP ;COMP;I;POS;ORDER
[1]     LF, 'CODE      PRODUCT                   COMPOSITION',LF
[2]     ORDER←⍋DERIVCO
[3]     L0: I←''⍴ORDER
[4]     POS←CODE⍳DERIVCO[I]
[5]     COMP←COMPOS[;I;]
[6]     COMP←(COMP[1;]≠0)/COMP
[7]     COMP← 6 0 ⍕ ⍉ COMP
[8]     COMP[;2]←'+'
[9]     COMP[;8]←'P'
[10]    (4 0⍕DERIVCO[I]),' ',PRODUCT[POS;],' ',2↓,COMP
[11]    →(0<⍴ORDER←1↓ORDER)/L0
      ∇
```

After transposition and formatting [7], we can insert a column of +'s and a column of *P*'s. Ravelling this matrix will give the vector of the compositions to be printed in [10].

The processing continues to exhaustion of the list of derivatives, by means of a loop [11].

Note: <u>*LF*</u> is a gloabal variable, which contains the line feed.

UPDATING THE LIST OF PRICES

ALL is the list of the only products whose price can be modified, i.e. those:

- which exist (information given by *BINPRO*),
- which are not derivatives: *CODE DERIVCO*

Instructions [2 and 3] serve to introduce the list of products whose price will be introduced. Of course, they must belong to the list above, otherwise an error message is emitted [4-5].

This check having been made, we can calculate the position *POS* of these products in the different data [6].

```
        ∇ UPDPRICE ;ALL;BIN;POS;SAME;CP
[1]     ALL←(BINPRO∧~CODE∊DERIVCO)/CODE
[2]     L1:LF,'WHICH PRODUCTS [ALL, END]'
[3]     →(END∊CP←⎕)/0
[4]     →(∊/BIN←CP ALL)/L2
[5]     →L1 AFTER 'NOT FITTING : ',⍕(~BIN)/CP
[6]     L2: POS←CODE⍳CP
[7]     L3: P←''⍴POS
[8]     SAME←PRICE[P]
[9]     PRODUCT[P;]
[10]    PRICE[P]←⌊''⍴⎕
[11]    →(0<⍴POS←1↓POS)/L3
[12]    CALPRICE
        ∇
```

Thus, product by product [7]:

- we look for its price, which is called *SAME* (instruction [8]),
- we display its name [9]
- we collect its new price, which directly updates the price vector [10].

A loop serves to survey all the products chosen [11]. The function *CALPRICE* then calculates the prices of derivatives.

ERASING A PRODUCT

Erasing a product is performed simply by introducing a zero in the correct place in *BINPRO*.[4]. We check the code and the product name beforehand by means of the *VERIPROD* function, already used [1].

We also check that the product to be erased is not used in the composition of other products [2-3]. If this were the case, an anomaly message would be printed [6 and 7].

```
        ∇ ERASEPROD ;PRO;DERIVE
[1]     L0: →(0=PRO←VERIPROD)/L2
[2]     DERIVE←(COMPOS[2;;]∨.=PRO)/DERIVCO
[3]     →(0<⍴DERIVE)/L1
[4]     BINPRO[CODE⍳PRO]←0
[5]     →L0
[6]     L1: 'THIS PRODUCT IS A COMPONENT OF : ',⍕DERIVE
[7]     →L0 AFTER 'IT MAY NOT BE ERASED'
[8]     L2: '----> END'
        ∇
```

CHANGE OF STATE

This process is very similar:

- checking the code and name by *VERIPROD*,
- if the product is not a derivative, print-out of an anomaly message [2-3],
- if not, *BIN* is a binary vector which serves to erase the product of *DERIVCO* and *COMPOS* (instructions [5 and 6]).

We can therefore update the price, possibly by reference to the old one [7 to 10].

Here again, *CALPRICE* solves to correct the prices altered by these modifications.

```
        ∇ ERASECOMP ;PRO;BIN;SAME;POS
[1]       L0: →(0=PRO←VERIPROD)/L2
[2]       →(PRO∊DERIVCO)/L1
[3]       →L0 AFTER 'THIS PRODUCT IS NOT A COMPOUND'
[4]       L1:
[5]       DERIVCO←(BIN←DERIVCO≠PRO)/DERIVCO
[6]       COMPOS←BIN/[2] COMPOS
[7]       SAME←PRICE[POS←CODE⍳PRO]
[8]       'NEW PRICE  [SAME]'
[9]       PRICE[POS]←⍎''⍴⎕
[10]      →L0
[11]      L2: CALPRICE
[12]      '----> END'
        ∇
```

CALCULATING PRICES

UNKNOWNS is the list of products whose price must be calculated. At the start, this is all the derivatives [1].

```
        ∇ CALPRICE ;UNKNOWNS;AGAIN
[1]       UNKNOWNS←DERIVCO
[2]       L0: →(0=AGAIN←1↑UNKNOWNS)/0
[3]       PRICE[CODE⍳AGAIN] ←  PRODPRICE AGAIN
[4]       →L0
        ∇
```

If this list becomes empty, *AGAIN* takes the value zero, and we leave the function. If not the *PRODPRICE* function calculates the current price of the product and updates the *PRICE* vector (instruction [3]).

Of course, in order to calculate this price, *PRODPRICE* has perhaps calculated other prices, and *UNKNOWNS* could have diminished accordingly. This *PRODPRICE* function constitutes the most delicate part.

PRODPRICE receives a code or a list of codes as operand. If they all have known prices, it suffices to consult the vector *PRICE*, to receive the appropriate prices as result [2 and 3].

If, on the contrary, some products feature in *UNKNOWNS*, we proceed with the calculation properly speaking:

- seeking the index of the product [4],
- extracting its components [5],
- eliminating the zero values in the components [6] and in quantities [7].

It is sufficient to multiply the quantities (in %) by the prices of the component products. However, as there is a risk that some prices are unknown, we use *PRODPRICE* to calculate them.

This funciton summons itself, it is said to be RECURSIVE.

Each time that the price of a product is known we erase its code from the list of unknowns [9].

```
      ∇ R←PRODPRICE P ;IP;BIN;QTES;COMPONENTS
[1]     →(P∈UNKNOWNS)/CALCUL
[2]     R←PRICE[CODE⍳P]
[3]     →0
[4]     CALCUL: IP←''⍴DERIVCO⍳P
[5]     BIN←0≠COMPONENTS←COMPOS[2;IP;]
[6]     COMPONENTS←BIN/COMPONENTS
[7]     QTES←BIN/COMPOS[1;IP;]
[8]     R←⌈0.01×QTES +.× PRODPRICE COMPONENTS
[9]     UNKNOWNS←(~UNKNOWNS∈P)/UNKNOWNS
      ∇
```

Successive calls for the function by itself end when all the components of a product are basic products, or they have been calculated during a preceding operation.

This way of solving the problem is perfectly logical, and simpler to program than methods employing the explicit arborescent search which mutually associates the products.

If you have done everything, well done!

THREE PUZZLES

THEME 1

```
        ∇ R←LONG JUSTIF TEXT   ;POS;I
[1]       POS ← (TEXT=' ')/I←ιρTEXT
[2]       I←I, (LONG-ρTEXT)ρPOS
[3]       R←TEXT[I[⍋I]]
        ∇
```

In [1] we look for the positions of the blanks in the text, whereas *I* receives the series of whole numbers from 1 to ρ*TEXT*.
For example, consider the vector '*A TEXT TO BE SEEN*'

POS equals 3 9 14

I is the series of whole numbers from 1 to 18.

In this way, *TEXT*[1] would give the text itself.

In [2] we supplement *I* with the positions of the blanks, to bring it to the required length. For example if *LENGTH* equals 25, we supplement *I* with 7ρ*POS* i.e:

3 9 14 3 9 14 3

In this way, *TEXT*[1] would give the initial text followed by the number of blanks necessary to give it the correct length. Re-arranging *I*, we obtain:

1 2 3 3 3 3 4 5 6 7 8 9 9 9 10 11 12 13 14 14 14 15 16 17 18

TEXT[*I*[⍋*I*]] thus enables the excessive blanks to be distributed at the positions of the existing blanks. We obtain the result below, indicating the blanks by the ‸ sign.

A‸‸‸‸TEXT‸‸‸TO‸‸‸BE‸‸‸SEEN

THEME 2

```
        ∇ R←N REPRO V ;⎕IO
[1]       ⎕IO←0
[2]       R←V[+\(ι+/N)∊+\N]
        ∇
```

The trick consists of creating a binary vector which has the length of the vector to be created, and which has a 1 at the beginning of each sentence. It is preferable to work in origin 0, whence the first instruction.

We will consider the example 3 2 4 *REPRO 'ROT'*

```
+\N   equals  3  5  9
⍳+/N  equals  0  1  2  3  4  5  6  7  8  because of the origin of
                                          the indices.
(⍳+/N)∊+\N  gives  0  0  0  1  0  1  0  0  0
a scan by the sum gives:  0  0  0  1  1  2  2  2  2
```

This is indeed the set of indices required to obtain the vector sought.

```
      'ROT'[0  0  0  1  1  2  2  2  2]
RRROOTTTT
```

This solution is not suitable if the left-hand argument contains zeros. Never mind! a minor modification would take us back to the preceding case:

```
      ∇ R←N REPRO V ;⎕IO
[1]     V←(N≠⎕IO←0)/V
[2]     R←V[+\(⍳+/N)∊+\(N≠0)/N]
      ∇
```

THEME 3

The problem consists of adding the scattered values in a matrix. We know that we can achieve this by simple indexing.
On the other hand, we can readily ravel the matrix *DIST* and index the vector thus obtained.

In order to ascertain the indices of the elements to be extracted we will convert the couple Origin→Destination by a method very similar to that of the *PLACEIN* function (see course, page 179).

```
      ∇ R ← D BUS T
[1]     T←(¯1↓T),[0.5](1↓T)
[2]     T←1+(⍴D) ⊥ T-1
[3]     R←+/(,D)[T]
      ∇
```

We start by creating a matrix whose first line contains the departure point of each simple journey, and the second contains the point of arrival for the journey 2 5 1 3 6, we obtain the matrix:

```
2  5  1  3
5  1  3  6
```

Hence each column constitutes the set of indices which enables the distance travelled to be extracted from the *DIST* matrix.

The second instruction gives the corresponding index in the vector ,*D*.

```
Hence,  2  5 becomes 11
        5  1 becomes 25
        1  3 becomes  3
        3  6 becomes 18
```

In the last line, (*D*,)[11 25 3 18] gives 10 6 7 3, the total of which gives the total distance travelled.

If we wish to be able to accept matrices of journeys, we must slightly modify the function:

- we must ensure that we have introduced a matrix in the right-hand argument. To this end, the first instruction has the effect of transforming a vector into a matrix.
- we must effect a shift on the columns of this matrix, which requires modification of the left-hand argument of the function ↓.

The remainder is unchanged:

```
      ∇ R←BUS T
[1]     T←(¯2↑ 1 1,⍴T)⍴T
[2]     T←(0 ¯1↓T),[0.5](0 1↓T)
[3]     T←1+(⍴D)⊥T-1
[4]     R←+/(,D)[T]
      ∇
```

TABLE OF FUNCTIONS

* indicates functions of general interest, likely to feature in an initial "kit of tools", after some possible adaptations. The page numbers refer to the answers.

* *A* 84
* *AFTER* 114
ARRAYS 130

* *BALANCE* 119
* *BETWEEN* 97
* *BLODIAG* 154
BUS 165

CALPRICE 162
* *CALTIME* 123
CALTRANS 141
* *CHARACTER* 143
COMPARE 102
* *CONDENSE* 118
CONSERVE 134
* *CROSS* 117
* *CROSSCOUNT* 140
* *CRUC* 117
CUT 127

DEFHEADER 144
DEFREST 129
DEFZONES 144
DISPLAY 88

ERASECLI 152
ERASECOMP 162
ERASEOUT 137
ERASEPROD 161
* *EXCEPT* 96
EXTRAPOLATE 113

FIND1 121
FIND2 122
FIND2B 122
FIND3 122
FIND4 123
FIND6 124
* *FORBLANK* 112

GETADD 149
GETCLI 183
GETGROUP 148
GETINA 148

I 114
* *IMBRICATES* 101
INIOUT 131
INSCRIPTIONS 133
INSTALMENT 134
INTROCLI 147
INTROPROD 158
INTROTEXT 98

JUSTIF 164

L 114
* *LIST* 106

MULTIBLO 123

PARTICIPANTS 137
* *PC* 85
PLAN 132
PREPARE 127
PRESENTATION 93
PRICL 150
PRINCOMP 160
PRINPROD 157
PRIREST 130
PRINT 103
PRINTCLI 150
* *PRINTEXT* 143
PRINT-OUT 92
PRINT-OUT 99
PRINV 94
PRODPRICE 163
PURGEOUT 138
PYRAMIDS 123

RECEIVES 86
* *REPRO* 164

* *SAVEWS* 132
SPLIT 91
STATE 136
STORE 149

TESTASS 104
* *TO* 83
* *TOTALISER* 117
TRANSFERS 141
TRIPS 135

UPDATE 139
UPCOMP 159
UPDPRICE 161

VERIPROD 159
* *VERT* 84

WELD 109
WELD 109
WELD2B 111
WELD 110
WELDIC 112
* *WITH* 85